UNIVERSITÉ DE MONTRÉAL

CONCEPTION ET MISE EN OEUVRE DE PROCESSEURS CONFIGURABLES POUR LA CONVERSION DE TAUX DE TRAMES VIDÉOS AVEC COMPENSATION DE MOUVEMENT

NICOLAS BEUCHER
DÉPARTEMENT DE GÉNIE ELECTRIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE ÉLECTRIQUE)
JUILLET 2007

Your file Votre référence
ISBN: 978-0-494-35665-4
Our file Notre référence
ISBN: 978-0-494-35665-4

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

CONCEPTION ET MISE EN OEUVRE DE PROCESSEURS CONFIGURABLES POUR LA CONVERSION DE TAUX DE TRAMES VIDÉOS AVEC COMPENSATION DE MOUVEMENT

présenté par: BEUCHER Nicolas

en vue de l'obtention du diplôme de: Maîtrise ès sciences appliquées

a été dûment accepté par le jury d'examen constitué de:

M. DAVID Jean-Pierre, Ph.D., président

M. SAVARIA Yvon, Ph.D., membre et directeur de recherche

M. BOIS Guy, Ph.D., membre et codirecteur de recherche

M. LANGLOIS Pierre, Ph.D., membre

À ma famille

REMERCIEMENTS

Plusieurs personnes ont permis la réalisation de ce travail tant par leur soutien technique qu'humain. Ils ont su prodiguer conseils et encouragements qui ont permis à mon travail de progresser rapidement dans la bonne direction.

En premier lieu un grand merci à Yvon Savaria pour avoir accepté de diriger ma maîtrise et m'avoir proposé un sujet aussi intéressant. Il a su donné de la valeur à mes travaux en les dirigeant sur le bon chemin et en offrant conseils et suggestions pour progresser.

Merci également à Guy Bois dont les conseils et la vision ont été d'excellentes sources d'inspiration. Ses précieuses relectures de mes articles ont également permis d'en améliorer la qualité.

Un remerciement tout particulier à Normand Bélanger qui par son support technique appuyé par ses grandes connaissances, sa disponibilité et sa gentillesse m'ont permis d'avancer à grands pas. Merci à lui aussi pour toutes les discussions légèrement hors-sujet mais si intéressantes ;-).

Merci également à Maria et Dany qui ont permis par leurs patientes explications de faire démarrer cette maîtrise dans les meilleures condiditions.

Enfin merci à Gilbert, Iliasse, Vincent, Rafael, Ali, Mohamed, Rahul et bien d'autres pour avoir fait de ces deux années au GRM un agréable séjour.

RÉSUMÉ

Ce mémoire présente les travaux réalisés pour accélérer l'implémentation d'algorithmes d'augmentation de taux de trames par compensation de mouvement (Motion Compensated Frame Rate Conversion - MC-FRC) à l'aide de jeux d'instructions spécialisés. Différentes classes existent parmis ces algorithmes ; les travaux présentés ici s'intéressent aux algorithmes basés sur la division en blocs des images. Deux algorithmes sont implementés afin de déterminer leurs caractéristiques. Le premier, considéré comme simple, n'est en fait que le noyau de ce type d'algorithme et il permet de se faire une bonne idée des enjeux découlant de leur utilisation. Le second est une version plus complexe qui fait appel à plusieurs techniques pour obtenir un meilleur résultat. Dans les deux cas, les algorithmes réclament une grande puissance de calcul. Le déroulement des algorithmes est donc accéléré à travers la conception de jeux d'instructions spécialisés. Ces derniers permettent des accélérations de plus de 100 fois pour l'algorithme simple et d'environ 50 fois pour l'algorithme complexe. Les travaux présentés s'intéressent aussi à l'efficacité énergétique des processeurs spécialisés. Une méthodologie a donc été mise en place afin de comparer la consommation d'énergie d'un processeur standard avec celle d'un processeur spécialisé. Les résultats basés sur l'algorithme simple montrent que le processeur étendu est énergétiquement 50 fois plus efficace que le processeur standard. Une extension de ces travaux montre qu'une règle simple, liant l'accélération de performance, le rapport des surfaces entre les deux processeurs et le ratio d'énergie entre ces derniers peut prédire ce résultat. Cette règle est validée par trois cas de test décrits dans ce mémoire. Enfin, la description de travaux permettant la mise en oeuvre de ce type de processeurs dans des environnements multiprocesseurs ainsi qu'une discussion sur les algorithmes d'augmentation de taux de trames par compensation de mouvement terminent ce document.

ABSTRACT

This document presents the work done to accelerate Motion Compensated Frame Rate Conversion (MC-FRC) algorithms using application specific instruction-sets. Many classes of MC-FRC algorithms exist ; here only the block motion estimation (BME) techniques are studied. Two algorithms are implemented to evaluate their characteristics. The first one, referred to as the simple one, is the main core of all MC-FRC algorithms based on BME. Its implementation gives a good idea of the work to do to accelerate such algorithms. The second algorithm, referred to as the complex one, is embedding more advanced techniques to increase the quality of the output. Both algorithms require a lot of computational effort and thus the need to develop specific instruction-sets to accelerate their execution. The acceleration reached range from more than 100-fold for the simple algorithm to about 50-fold for the complex one. The energy efficiency of extended processors is also of interest in the span of this work. Consequently, methodology to compare the energy consumption of the extended processor with the energy consumption of the standard processor has been built. The results show that, in the case of the simple algorithm, the extended processor is about 50 times more energy-efficient than the standard processor. An extension of this work also proves that this result could have been forecasted by a simple equation using the acceleration factor and the area overhead to deduce the energy saved. This simple model was validated by three test cases. To conclude, a description of the work done to allow simulation of these processors in more complex environnements (i.e. multiprocessors with network on chip systems) and a short discussion regarding MC-FRC algorithms close this document.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

LISTE DES NOTATIONS ET DES SYMBOLES

$ASIC$: Application Specific Integrated Circuit, circuit intégré dédié à une application spécifique

$ASIP$: Application Specific Instruction-set Processor, processor à jeu d'instructions spécialisé

$FPGA$: Field Programmable Gate Array, r éseau prédiffusé programmable

DSP : Digital Signal Processor, processeur de signal numérique

$MC-FRC$:Motion Compensated Frame Rate Conversion, augmentation de taux de trames par compensation de mouvement

BME : Block Motion Estimation, estimation du mouvement par bloc

BMA : Block Matching Algorithm, algorithme de correspondance de bloc

FS : Full Search, recherche complète

$ODFS$: One-Dimensional Full Search, recherche complète sur une dimension

HD : Haute-Définition

ALU : Arithmethic qnd Logic Unit, unité arithmétique et logique

$JPEG$: Joint Photographic Experts Group

INTRODUCTION

La vidéo constitue l'un des marchés les plus foisonnants de la micro-électronique de loisir à l'heure actuelle. Avec la banalisation des appareils vidéos portables, l'essor de la télévision haute définition (HD) et le succès des sites web de partage de vidéo, les applications vidéos sont de plus en plus courantes et la demande très importante. L'enjeu est de créer des systèmes performants, adaptables et économiques pour répondre à cette demande. Les problèmes sont nombreux et concernent des domaines très larges de micro-électronique, d'informatique et d'algorithmique. Un des grands enjeux est de pouvoir faire face à la multiplication des sources disponibles, bien souvent dans des formats différents. La taille de l'image, le taux de trames ou encore l'encodage utilisé sont autant de variables auxquelles il faut faire face. Cette diversité impose le développement de longue chaînes de traitement afin d'être capable de supporter un vaste nombre de situations. Dans le présent mémoire de maîtrise, on s'intéresse à la conversion du taux de trames.

Le taux de trames, ou débit d'images, est une caractéristique essentielle d'une vidéo. En général, ce taux est supérieur à 10 images par seconde afin d'obtenir l'illusion du mouvement. En deçà de ce seuil, l'oeil humain est capable de discerner la séquence des images. En réalité pourtant, les vidéos auront tendance à utiliser des taux bien supérieurs (au moins le double) car ce taux est un élément important du confort de vision et les variations de perceptions entre individus imposent de s'assurer que l'on présente une bonne qualité pour tous. Seules les vidéos de basse qualité utilisent des taux de l'ordre de la dizaine. Au cinéma, le débit d'images est de 24 images par seconde. Les télévisions NTSC affichent 30 images par seconde. Dans ce dernier cas, cela s'explique historiquement par la fréquence des alimentations électriques en Amérique du Nord qui est fixée à 60 Hz. Les télévisions affichent les lignes paires puis les lignes impaires avec une fréquence de balayage vertical de 60 Hz. Les télévisions PAL, concernant l'Europe

notamment, utilisent un débit d'images de 25 par seconde pour des raisons similaires. Le taux de trames peut également prendre des valeurs bien plus élevées, jusqu'à 100, dans le cas de la télévision HD. Ceci impose donc d'être capable de convertir les différentes normes entre elles. C'est la tâche des algorithmes de conversion de taux de trames. Il existe plusieurs types d'algorithmes pour réaliser cette conversion, mais, dans le cadre de ce projet, seuls les algorithmes à compensation de mouvement seront étudiés (Motion Compensated Frame Rate Conversion - MC-FRC). Ils sont réputés offrir la meilleure qualité d'image au prix cependant d'efforts de calcul importants.

Une conséquence directe du coût en calcul de ces algorithmes est qu'ils réclament d'importants moyens matériels pour être mis en oeuvre. La technologie des processeurs configurables, aussi appelés processeurs à jeu d'instructions spécialisé (Application Specific Instruction-set Processor - ASIP), présente de nombreux intérêts pour ce type d'applications. Elle est rapide et aisée à utiliser et ne requièrt pas autant d'efforts que la conception d'une puce spécialisée. Elle peut offrir des performances suffisantes pour faire les traitements requis. Enfin, sa grande réutilisabilté et son adaptativité sont des atouts certains pour ce type d'application puisqu'on peut ainsi concevoir une puce générique qui pourra être utilisée aussi bien dans des applications embarquées que sur des systèmes fixes.

Les travaux présentés dans ce mémoire portent sur la mise en oeuvre de la technologie des processeurs configurables dans le cadre des algorithmes d'augmentation de taux de trames par compensation de mouvement. Les travaux s'intéressent particulièrement à la création d'un jeu d'instructions spécialisé pour ce type d'application en vue d'augmenter la performance. Deux algorithmes issus de la littérature ont été implantés afin de servir de base de départ. Les jeux d'instructions créés permettent d'augmenter significativement la performance par des facteurs pouvant aller jusqu'à plus de 100 fois. Les travaux s'intéressent aussi à la consommation d'énergie engendrée par l'utilisation d'instructions spécifiques. Les résultats montrent que la spécialisation diminue la

consommation énergétique dans le même ordre de grandeur que l'accélération. Enfin, une partie des travaux porte sur le développement d'une architecture centrée autour d'un processeur spécialisé pour mettre en oeuvre l'algorithme de façon efficace.

Le présent mémoire est constitué de 7 chapitres en plus de l'introduction. Le chapitre 1 présente en détails la technologie des processeurs spécialisés (ASIP). Les forces et les faiblesses de cette technologie sont présentées ainsi que la méthodologie associée à leur utilisation. Le chapitre 2 présente les algorithmes de MC-FRC de façon générale. Leur intérêt et leurs particularités sont discutés. Ce chapitre présente également les deux algorithmes retenus pour ce projet. Le chapitre 3 présente les jeux d'instructions spécialisés créés pour accélérer le traitement des algorithmes de MC-FRC retenus. Les instructions sont décrites et expliquées. Le chapitre présente également les méthodes utilisées pour la vérification des instructions ainsi que les modifications apportées aux programmes C pour que ces derniers utilisent le nouveau jeu d'instructions. Le chapitre 4 présente les résultats obtenus avec les nouveaux jeux d'instructions et notamment l'accélération qu'ils ont permis. Ce chapitre discute également ces résultats. Le chapitre 5 présente la méthodologie développée pour estimer la consommation de puissance du jeu d'instructions étendu. Les résultats obtenus sont également présentés et discutés. Ce chapitre présente également une discussion sur le rapport entre l'énergie, l'accélération et la superficie. Deux cas, différents du MC-FRC, sont présentés. Le chapitre 6 présente les travaux annexes à ce projet et les extensions futures. Les travaux concernant l'interconnexion entre un processeur spécialisé et un réseau sur puce sont notamment présentés dans ce chapitre. Le dernier chapitre conclut ce mémoire.

CHAPITRE 1

TECHNOLOGIE DES ASIP : INTÉRÊT ET POSSIBILITÉS

1.1 Généralités

Le terme ASIP, qui signifie Application Specific Instruction-set Processor, est utilisé pour désigner un grand nombre de processeurs ; les processeurs spécialisés qui possèdent un jeu d'instructions très spécifique qui les rend capables de réaliser des opérations particulières (par exemple les DSP), les processeurs configurables dont on peut modifier les caractéristiques internes comme la quantité de registres ou la nature de la mémoire cache (comme, par exemple, Nios de Altera) et les processeurs standards pour lesquels on peut aisément ajouter un jeu d'instruction supplémentaire dits processeurs extensibles(Xtensa de Tensilica[27]).

Les ASIP conservent la programmabilité des processeurs standards et permettent d'augmenter les performances de façon suffisante pour rivaliser avec des ASIC (Application Specific Integrated Circuits). Lorsqu'on les compare avec ces derniers, les ASIP sont plus faciles à mettre en oeuvre, à vérifier et finalement à réutiliser que les ASIC mais ils ne sont pas toujours suffisamment performants et sont bien moins efficaces en terme d'énergie qu'une puce spécialisée. Comparés aux processeurs standards, ils sont par contre bien plus performants (pour une application donnée) et consomment moins. Il perdent toutefois en partie la généralité de ces derniers car leur jeu d'instructions impose des contraintes plus fortes.

Toutefois, dans le cas des processeurs extensibles (Xtensa), on conserve la généralité du processeur standard tout en accédant à la performance d'un ASIP simple. Cela

implique toutefois un processeur plus grand en superficie et sans doute moins efficace en termes d'énergie, mais cette voie peut être une bonne solution pour les systèmes électroniques complexes qui peuvent embarquer plusieurs processeurs et structures spécialisées au sein d'une même puce. De tels systèmes sont relativement fréquents de nos jours. Un téléphone cellulaire classique possède généralement plusieurs DSP, pour le traitement de la voix et du signal, un processeur standard qui se charge de l'interface avec l'utilisateur et plusieurs autres éléments selon les capacités de l'appareil (vidéo, audio MP3, etc ...). Les ASIP basés sur l'extension peuvent alors être utiles pour remplacer efficacement plusieurs de ces modules tout en conservant la puissance de calcul nécessaire. Ils réduisent ainsi la complexité du système. Malgré tout les processeurs extensibles sont limités par leur base qui restreint les possibilités du concepteur. Certains ASIP peuvent être créés à partir de rien évitant ainsi le surplus dû à la base standard présente dans les processeurs extensibles. Par exemple, la technologie Lisatek de la compagnie Coware [10] permet de créer l'intégralité du processeur selon les besoins en performance. Ce type de technologie est donc plus souple que la technologie des processeurs extensibles mais elle exige davantage d'efforts de la part des concepteurs.

Leur grande généralité ainsi que leur performance générale a permis aux ASIP d'être utilisés dans de nombreux domaines tels que les traitements vidéos [18, 20, 22], le traitement du signal [1] ou encore la conception de puces pour les réseaux [23].

Des travaux complémentaires se consacrent à l'automatisation de la création des jeux d'instructions spécifiques. Le but étant de développer des outils qui analysent le code source de l'application, cherchent les goulots d'étranglement et créent les instructions permettant d'accélérer le traitement sans intervention humaine [2, 25].

1.2 Xtensa LX

Le Xtensa LX[28] est un processeur produit par la compagnie Tensilica. C'est un processeur configurable auquel on peut rajouter un jeu d'instructions spécialisé.

Le concepteur utilisant le Xtensa LX a donc la possibilité de créer un processeur standard selon ses spécifications en termes de performance, de superficie et de consommation énergétique. S'il souhaite obtenir plus de performance, il a également la possibilité d'ajouter, en suivant certaines règles, un jeu d'instructions supplémentaire qui ciblera un éventuel goulot d'étranglement de son algorithme.

Plus spécifiquement, la compagnie Tensilica met à la disposition du concepteur une interface de création de processeur, XPG (Xtensa Processor Generator), qui permet de créer et de configurer un processeur basé sur leur architecture de processeur 32 bits, Xtensa LX. Le concepteur peut modifier un certain nombre de paramètres internes comme l'ajout de certaines instructions (multiplieur 32 bits ou MAC), le nombre de registres, l'ordre mémoire des octets (gros ou petit boutiste (endian)), ou encore la taille et l'associativité des caches d'instructions et de données. Il peut également, s'il le souhaite, attacher directement au processeur des mémoires RAM ou ROM. L'outil XPG affiche tout au long de ses choix des estimations sur la puissance consommée, la superficie et la fréquence d'horloge maximale à laquelle le processeur sera capable de fonctionner.

Lorsque le concepteur est sûr de ses choix, il peut télécharger une gamme d'outils ciblés pour le processeur créé. Ces outils contiennent entre autres un compilateur C (xt-xcc), un simulateur (xt-run) et un outil de profilage (xt-gprof). Ils permettent de compiler et de simuler un programme C et d'obtenir des statistiques de performance. Le simulateur retourne le nombre exact de cycles qu'il a fallu pour compléter le programme ainsi que des statistiques sur l'utilisation de la cache. Le profileur permet de détailler les résultats obtenus pour chaque fonction du programme afin de cerner les goulots d'étran-

glement de l'application.

Si la performance souhaitée n'est pas atteinte, le concepteur peut alors, à l'aide d'une autre gamme d'outils, créer et intégrer un jeu d'instructions spécialisé pour accélérer le traitement des points chauds de l'algorithme. La création d'instructions spécialisées se fait à l'aide du langage TIE (Tensilica Instruction Extension). Le langage TIE est très proche du Verilog. Un certain nombre de mots clés supplémentaires permettent de créer de nouveaux fichiers de registres, des registres d'état ou encore des instructions spécifiques. Une fois le fichier TIE créé, il doit ensuite être compilé avec un compilateur spécial (tc) qui retourne un certain nombre de fichiers résultats. En premier lieu, le compilateur retourne un fichier Verilog décrivant les nouvelles instructions. Ce fichier peut être attaché aux fichiers HDL décrivant le processeur standard. Les nouvelles instructions sont donc directement intégrées au chemin de données du processeur. Le compilateur retourne également des scripts pour synthétiser les instructions spécialisées. Ceci permet d'obtenir la superficie supplémentaire ajoutée par le jeu d'instructions étendu et aussi de connaître la fréquence maximale de fonctionnement autorisée par ce dernier. Enfin, le compilateur retourne un certain nombre de fichiers qui permettent au concepteur d'utiliser ce nouveau jeu d'instructions dans son programme, de le compiler et de le simuler en tenant compte des nouvelles instructions.

Comme on peut le voir, les outils fournis par Tensilica permettent au concepteur de réaliser un processeur spécialisé et de le tester de façon relativement simple et rapide.

Les outils utilisés pour établir les résultats présentés dans ce document utilisent tous la version RA2006.6. Le processeur de base utilisé est donc un Xtensa LX qui possède les instructions standards ainsi qu'un multiplieur 32-bit. Ce processeur embarque également une cache de données de 2 Ko configurée en ensemble associatif à 2 voies ainsi qu'un cache d'instruction de 1 Ko à accès direct. Le processeur peut communiquer avec des composants externes (bus, mémoires, DMA...) via une interface supportant des

mots de 128 bits. Ce processeur requiert un équivalent de quelques 64348 portes en superficie et fonctionne à une fréquence de 268 MHz d'après les indications fournies par l'outil de création XPG.

1.3 Méthodologie pour la création d'un jeu d'instructions spécialisé

Créer un jeu d'instructions spécialisé se fait par étapes successives. La création s'appuie sur une implantation logicielle de l'algorithme ou de l'application que l'on souhaite accélérer à l'aide d'instructions spécifiques. Cette implantation est créée sur un ordinateur classique et optimisée pour être la plus rapide et la plus efficace possible. On peut ensuite profiler l'application sur le processeur embarqué à l'aide d'outils comme un simulateur et un profileur. Cela permet de déterminer les goulots d'étranglement de l'application. Cette version logicielle va donc servir de référence fonctionnelle mais aussi de référence de performance pour évaluer l'impact des nouvelles instructions sur les goulots d'étranglement. La création du nouveau jeu d'instructions aura donc pour but de réduire ces goulots. Chaque instruction créée doit être vérifiée afin de s'assurer qu'elle remplit correctement sa tâche. Une fois qu'un premier exemplaire du jeu d'instructions est créé et fonctionnel, le concepteur doit s'assurer de ses performances en réalisant un nouveau profilage de l'application, qui cette fois-ci, utilise le jeu étendu. Si la performance atteinte n'est pas suffisante, le concepteur doit améliorer le jeu d'instructions créé, en concevant des instructions supplémentaires ou en optimisant celles qui existent déjà, et répéter ce processus autant de fois que nécessaire afin d'ateindre le but fixé. Lorsque la performance atteinte est satisfaisante, le concepteur doit vérifier que l'ajout du nouveau jeu d'instructions remplit bien les contraintes temporelles (fréquences de l'horloge) et de superficie fixées. Là encore, si ces contraintes ne sont pas rencontrées, le concepteur doit retourner à sa table à dessin afin de modifier les instructions pour qu'elles puissent rencontrer ces contraintes. Bien entendu, chaque modification doit être faite en vérifiant

que la performance nécessaire est toujours atteinte. Le processus est ainsi répété jusqu'à ce que le jeu d'instructions créé satisfasse toutes les contraintes : performance, superficie, fréquence d'horloge. La figure 1.1 schématise ce processus. C'est le processus qui a été appliqué au cours de ces travaux pour créer les diverses instructions spécialisées.

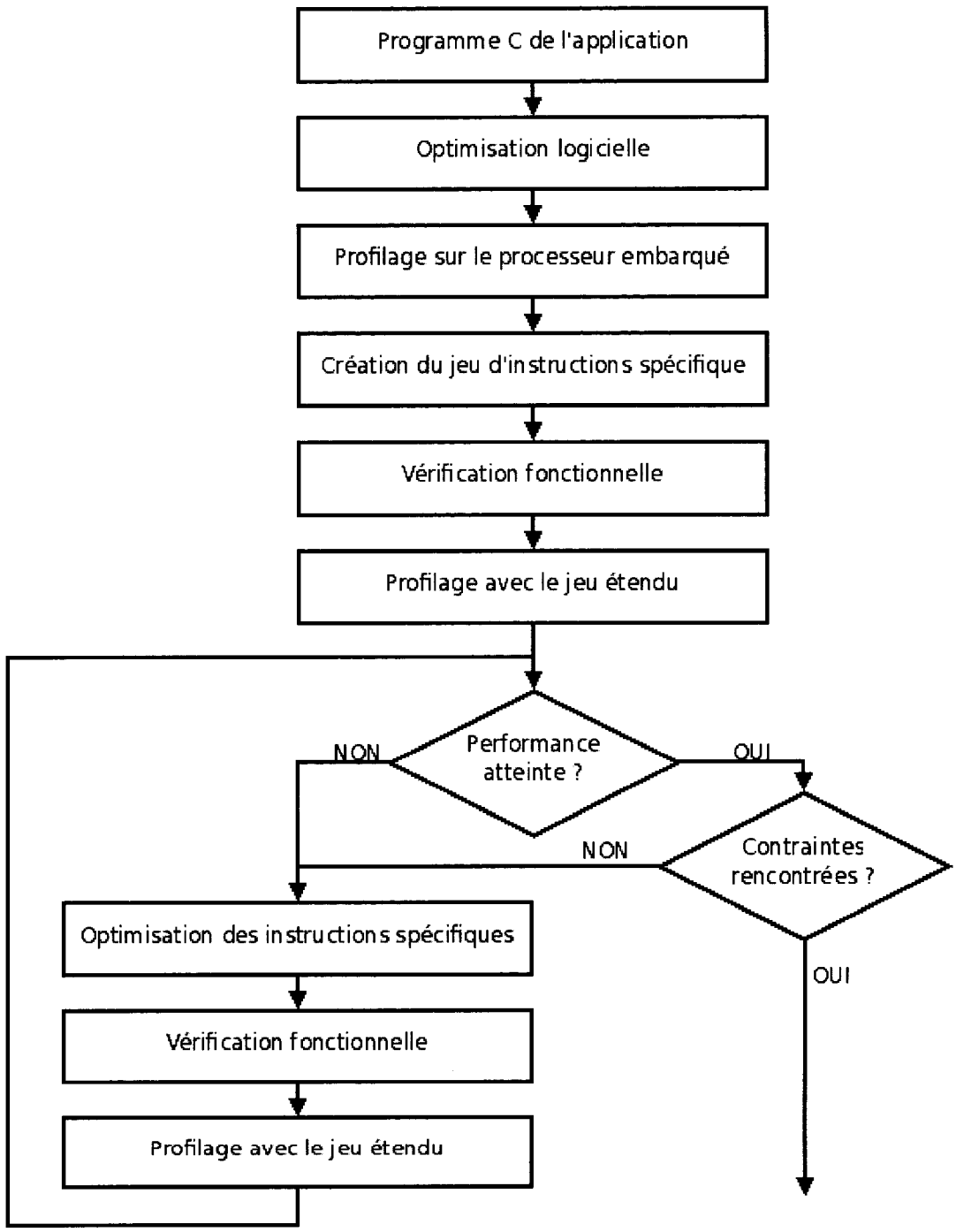


FIG. 1.1 Méthodologie de création d'un jeu d'instruction spécifique

CHAPITRE 2

ALGORITHMES DE MC-FRC

Dans ce chapitre, le principe des algorithmes d'augmentation de taux de trames avec compensation de mouvement (Motion Compensated Frame Rate Conversion - MC-FRC) est présenté. Ce chapitre présente également les deux algorithmes utilisés pour établir les résultats présentés dans ce document.

2.1 Revue de littérature

Les algorithmes de Frame Rate Conversion (FRC), parfois appelés Frame Rate-up Conversion, sont utilisés pour augmenter le taux d'images par seconde (frame per second - fps) d'un flux vidéo.

Tous les types de flux vidéo possèdent un débit d'images propre. Ainsi, dans les salles de cinéma, on diffuse les films avec un taux d'images par seconde de 24. Sur les télévisions nord-américaines, qui suivent la norme NTSC, ce taux est de 30 images par seconde alors que les télévisions européennes fonctionnent sur du 25 images par seconde. Le problème se complique encore si l'on considère les flux vidéo internet, la vidéo mobile (téléphone cellulaire et autres ipod vidéo) ou encore la télévision Haute Définition HD) qui étendent les possibilités en matière de taux d'images. L'existence d'un grand nombre possible de taux de trames impose la nécessité d'un algorithme de conversion. Le problème ne se posant réellement que dans le cas de l'augmentation du taux qui est bien plus complexe à réaliser que la simple diminution souvent réalisée en retirant des images du flux.

Il existe un certain nombre d'algorithmes pour réaliser cette conversion qui fournissent des résultats de qualité et de simplicité variables. L'exemple de la conversion cinéma vers télévision est assez frappant. Pour la conversion du cinéma vers les téléviseurs européens, le choix qui a été fait a été de simplement augmenter la vitesse du film. Ainsi le film passe 25 images par seconde au lieu des 24 initiales. La durée du film est donc réduite lorsque le film est diffusé à la télévision. Pour les téléviseurs nord-américain on utilise l'algorithme de 3 :2 pulldown [15]. Cet algorithme permet de passer de 24 images par seconde à 30 images par secondes, il lui faut donc générer 6 images par secondes pour compléter le flux vidéo originel. Cette augmentation revient donc à créer une cinquième image pour chaque lot de 4 images d'origine. L'algorithme de 3 :2 pulldown réalise cette tâche en entrelaçant les lignes paires de la troisième image d'origine avec les lignes impaires de la quatrième pour former une cinquième image qui sera diffusée entre les troisième et quatrième d'origine. Cet algorithme est très efficace pour les télévisions car il ne nécessite pas de calculs et il s'appuie sur le fonctionnement des téléviseurs qui affichent les lignes paires puis les lignes impaires mais il ne convient qu'à un seul et unique type de conversion. Il existe des algorithmes qui permettent de convertir n'importe quel débit d'image en un autre.

Parmi ces algorithmes, on compte les algorithmes à répétition d'images, les algorithmes à interpolation temporelle et les algorithmes à compensation de mouvement. La répétition d'image est la méthode la plus simple pour compléter un flux vidéo dans lequel il manque des images. Ainsi, pour passer d'un flux de 24 images par seconde à un flux de 30 images par seconde, on répéterait la dernière image de chaque lot de 4 pour en créer une cinquième. Cette méthode est très simple et ne nécessite aucun calcul mais elle dégrade la qualité de la vidéo en introduisant une impression de mouvement saccadé peu agréable à l'oeil. L'interpolation temporelle consiste à moyenner dans le temps les images originelles afin de créer les images manquantes. Bien que cette méthode ne produise pas de mouvements saccadés, elle introduit du flou dans l'image qui peut s'avé-

rer gênant pour le spectateur. Comme on le voit, ces deux méthodes sont simples mais elles ne répondent pas aux exigences en matière de qualité d'image et de comfort visuel. Sur des téléviseurs HD, il est important de disposer de méthodes plus performantes qui maintiennent un haut niveau de qualité.

Les algorithmes à compensation de mouvement (Motion Compensated Frame Rate Conversion - MC-FRC) sont réputés offrir une bonne qualité d'image[6, 16, 9]. L'idée est d'analyser les mouvements entre les images au sein du flux vidéo et d'en déduire l'image qu'il faut ajouter pour augmenter le taux d'images par seconde sans briser la chaîne du mouvement, ce qui donnerait l'impression d'une saccade, et en évitant d'introduire du flou. L'algorithme est donc mathématiquement plus complexe puisqu'il faut d'abord calculer et estimer les mouvements au sein de l'image. Pour ce faire, de multiples méthodes existent [19, 11], parmi lesquelles on compte : l'estimation de mouvement global (Global Motion Estimation - GME), l'estimation de mouvement de blocs (Block Motion Estimation - BME), l'estimation de mouvement de blocs avec recouvrement (Overlapped Block Motion Estimation - OBME) [14], l'estimation par gradient et l'estimation par récursivité (pel-recursive). L'estimation de mouvement global permet de suivre les mouvements larges de la caméra mais est complètement aveugle aux objets se déplaçant à l'intérieur du champ visuel, ce qui la rend très mauvaise pour réaliser du MC-FRC. Toutefois, il peut être utile de connaître le mouvement général de la caméra pour augmenter la qualité des résultats. L'estimation par mouvement de blocs consiste à découper les images en blocs et à analyser le mouvement de chacun d'entre eux pour obtenir un champ de vecteurs de déplacement au sein de l'image. Ces méthodes proviennent du domaine de la compression vidéo (famille des encodeurs vidéos MPEG) où elles ont été utilisées avec succès. Toutefois, ces méthodes possèdent l'inconvénient de ne pas capter tous les types de mouvements (notamment la dilatation et la rotation qui, du fait de la nature rigide des blocs, ne sont pas détectables). De plus, elles présentent des défauts dits *d'artéfacts de blocs* qui surviennent lorsque le mouvement estimé d'un bloc

ne correspond pas au mouvement réel, le bloc est alors déplacé de façon incorrecte et sa position provoque une discontinuité dans l'image. Ces discontinuités sont très visibles pour l'observateur car elles présentent un gradient de luminosité élevé auquel l'oeil est très sensible. Pour remédier à ce problème, des techniques d'estimation de mouvement de bloc avec recouvrement ont été suggérées. Ces techniques utilisent des blocs plus larges qui se recouvrent les uns sur les autres. Cela permet de corriger les problèmes de discontinuité car les parties qui se recouvrent sont moyennées, atténuant ainsi les erreurs dues au mauvais déplacement d'un bloc. Elles présentent toutefois le même inconvénient que la simple BME car elles ne permettent pas de détecter des rotations ou des dilatations. Les techniques de gradient et de récursivité travaillent pixel par pixel et cherchent à trouver le mouvement par des techniques de descente de gradient sur la luminosité d'un pixel. Ces techniques sont encore plus coûteuse que les techniques de blocs car elles travaillent pixel par pixel et nécessitent plusieurs passes par pixel pour obtenir un résultat et elles sont également plus sensibles au bruit sur l'image. Toutefois, elles n'ont pas de limitations intrinsèques sur le type de mouvement dans l'image.

Comme on le voit, il existe plusieurs méthodes pour réaliser l'estimation de mouvement. Elles ne possèdent pas toutes les mêmes caractéristiques et ont chacune leurs avantages et désavantages. Les méthodes les plus couramment observées dans la littérature se basent sur des méthodes de BME. En premier lieu, ces méthodes sont simples à comprendre et à mettre en oeuvre. De plus, elles sont bien connues car exploitées dans l'encodage vidéo. Les diverses recherches ont conduit à la création de multiples techniques visant à réduire l'effort de calcul et à augmenter la qualité de l'estimation. Toutefois, les techniques de BME sont toutes basées sur les mêmes principes. La première étape consiste à diviser l'image courante en blocs d'une certaine taille, dans certains cas, ces blocs sont contigus les uns aux autres et dans d'autres cas ils se juxtaposent. Afin de déterminer le mouvement suivi par un bloc, on cherche à déterminer sa position dans une ou plusieurs images précédentes. Cette recherche est effectuée par des algorithmes

de correspondance de blocs (Block-matching algorithms - BMA). Afin de ne pas parcourir inutilement l'intégralité de l'image précédente, on limite la recherche à une portion de cette image centrée autour de la position du bloc dans l'image courante. Dans cet espace, on cherche la position qu'occupait le bloc dans l'image précédente en le comparant avec un certain nombre de blocs de cet espace. La comparaison, qui est en réalité une distance mathématique sur la valeurs des pixels des deux blocs considérés, est une simple somme des valeurs absolues des différences des pixels des blocs (Sum of Absolute Difference - SAD). L'équation 2.1 montre le calcul de la SAD pour un bloc de taille N situé à la position (x,y) dans l'image courante (t) avec le bloc situé à la position (x,y) décalée de (u,v) dans l'image précédente (t-1). F représente la valeur du pixel à la position considérée.

$$SAD_{(x,y)}(u,v) = \sum_{i=0}^{N}\sum_{j=0}^{N} |F_t(x+i, y+j) - F_{t-1}(x+i+u, y+j+v)| \qquad (2.1)$$

L'ancienne position du bloc est considérée trouvée lorsque la somme rencontre son minimum. Il peut bien sûr exister plusieurs minimums, dans ce cas seul le premier sera conservé comme la position précédente. Avec l'ancienne position et la position actuelle du bloc, on peut calculer un vecteur de déplacement qui représente l'estimation du mouvement.

Le parcours de l'espace de recherche peut se faire selon diverses méthodes [31]. La plus simple et la plus gourmande en calcul est appelée Full Search (FS) ou encore Two-Dimensional Full Search (2DFS). Elle consiste à calculer la SAD pour tous les blocs de l'espace de recherche. Avec cette méthode, on est donc certain de trouver le minimum dans l'espace de recherche. Une autre méthode parcourt l'espace de recherche selon une dimension à la fois. Cette méthode, appelée One-Dimensional Full Search (ODFS) [8], teste d'abord toutes les blocs sur la ligne puis, à partir de la position du minimum sur

cette ligne, teste la colonne correspondante. Ce processus est répété une seconde fois sur un espace de recherche plus petit centré sur la position minimum trouvée lors de la première passe. Avec cette méthode, on réduit le nombre de calcul de SAD à effectuer. Il existe d'autres méthodes telles que le Three-Step-Search[7] ou encore le Four-Step-Search[26]. Ces méthodes utilisent toutes une procédure de recherche particulière pour trouver le minimum à l'intérieur de l'espace de recherche sans avoir à le parcourir en intégralité. Toutes ces méthodes permettent de réduire le coût de calculs de l'algorithme de BME.

Pour conclure sur cette revue de littérature, il convient de noter ici un point important. Jusqu'ici nous avons considéré que l'augmentation du débit d'image consistait à simplement générer le bon nombre d'images supplémentaires pour compléter un flux vidéo. En théorie, la conversion d'un flux vidéo en un autre est plus complexe que le simple ajout d'un nombre fixé d'images. Dans un flux vidéo normal, chaque image est séparée de la suivante par un temps fixe. Au cinéma par exemple, il s'écoule 1/24 de seconde entre deux images. Les mouvements que l'on voit à l'écran sont donc le résultat de l'intégration de la vitesse des objets durant ce laps de temps. En reprenant la séquence de quatre images issues d'un film de cinéma et en considérant qu'on lui ajoute afin d'être en mesure de diffuser le film à la télévision une cinquième image, si l'on regarde les temps des mouvements entre les images on observe ceci : de 1 à 2 les mouvements correspondent à 1/24 de seconde, de même pour 2 vers 3 et pour 3 vers 4. Entre 4 et l'image créée, la 5, le temps du mouvement n'est de 1/48 et il est également de 1/48 entre cette image et la 6. Les temps de diffusion sont par contre de 1/30 entre chaque image. Les temps de diffusion ne correspondent donc plus aux temps des mouvements sur les images. Il faudrait alors recréer intégralement la séquence d'image pour faire correspondre les deux temps (diffusion et mouvement). Or, cela requiert d'importants efforts de calculs. De nombreux travaux [6, 15] se contentent de créer une cinquième image et de distordre les temps en essayant toutefois de limiter au maximum l'impact

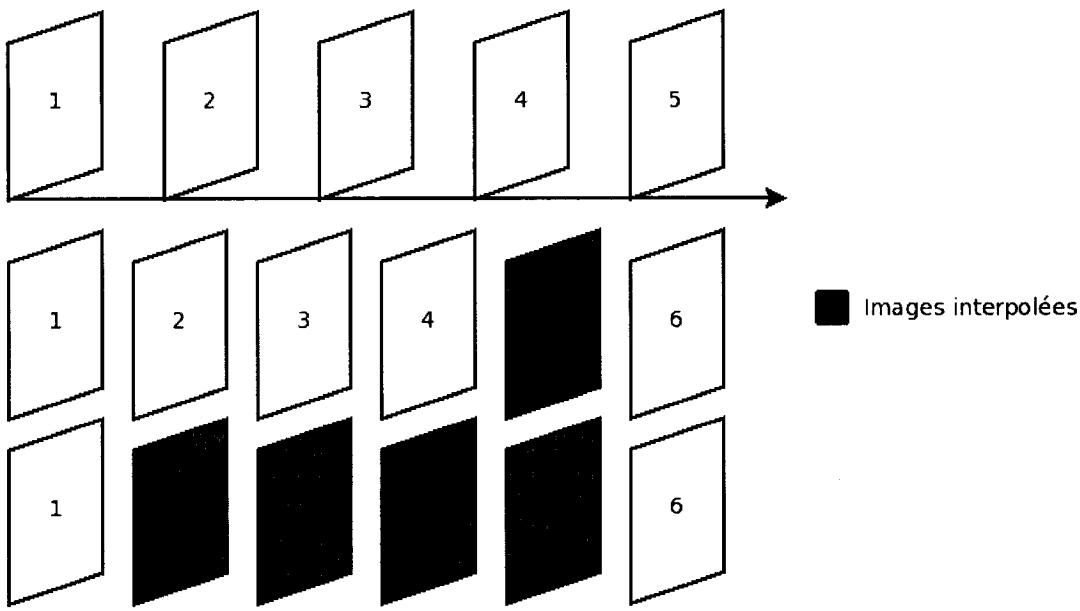


FIG. 2.1 Conversion de débit de 24 fps vers 30 fps, deux solutions possibles

visuel de cette distortion. La figure 2.1 montre les deux solutions possibles pour convertir le débit d'un film de cinéma vers un film adapté à une diffusion télévisée (NTSC). La première solution présente de nombreux avantages :

- On conserve les données d'origine, lesquelles sont en général de meilleures qualité que ce que peut faire un algorithme de MC-FRC.
- L'effort de calcul est moins important puisqu'il ne faut créer qu'une seule image (au lieu de trois dans l'exemple).
- L'interpolation du mouvement est plus simple à implanter en matériel car elle s'obtient par une division par deux. On réduit ainsi la logique de calcul.

2.2 Algorithme simple

L'algorithme simple se base sur de l'estimation de mouvements de blocs. Il travaille avec 2 images référencées comme l'image courante et l'image précédente. L'image courante est divisée en blocs carrés de taille n (n peut prendre des valeurs entières variables mais qui doivent toutefois être des diviseurs de la hauteur et de la largeur de l'image). Les blocs sont contigus. Ensuite, pour chaque bloc de l'image, on cherche la position qu'il occupait dans l'image précédente à l'intérieur d'un espace de recherche de taille m centré autour de la position du bloc dans l'image courante. Le paramètre m indique que l'on va pouvoir tester toutes les positions allant de (-m,-m) à (m,m) relativement à la position du bloc considéré. La recherche dans cet espace peut se faire selon la méthode de Full Search (FS) ou selon ODFS, évoquées en 2.1. Une fois l'espace de recherche parcouru, on obtient la position estimée du bloc dans l'image précédente qui nous permet de calculer le vecteur de déplacement estimé. Ce vecteur est ensuite divisé par 2, puisqu'on cherche à générer une image située au milieu des images courante et précédente, pour obtenir la position qu'aurait le bloc dans l'image interpolée. Le bloc est ensuite déplacé selon ce vecteur pour générer l'image interpolée. La figure 2.2 présente un schéma du fonctionnement de cet algorithme. Un algorithme aussi simple ne donne pas des résultats de très grande qualité et, notamment, il arrive souvent que des pixels à l'intérieur de l'image interpolée soient orphelins, i.e s'ils n'ont pas été recouverts par un bloc déplacé. Ils restent donc nuls et se caractérisent par des rectangles noirs de tailles diverses dans l'image. Ces rectangles sont très visibles. Une correction simple consiste à copier à la place de ces pixels les valeurs des pixels à la même position dans l'image courante. Toutefois, même ainsi, le nombre d'artéfacts de blocs reste important et ces derniers sont visibles.

Les résultats de cet algorithme, dont on peut avoir un aperçu dans la figure 2.3, peuvent être améliorés de façon significative en procédant à plusieurs passes. Dans un

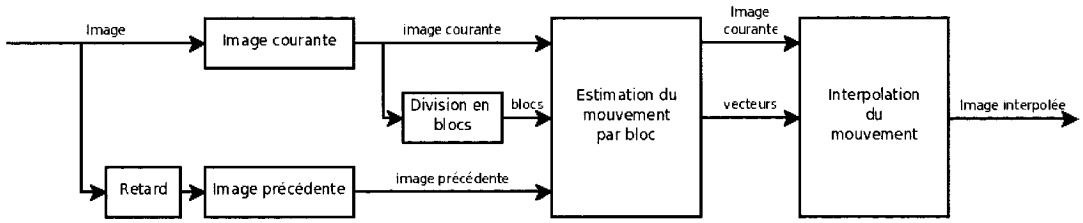


FIG. 2.2 Schéma de l'algorithme simple

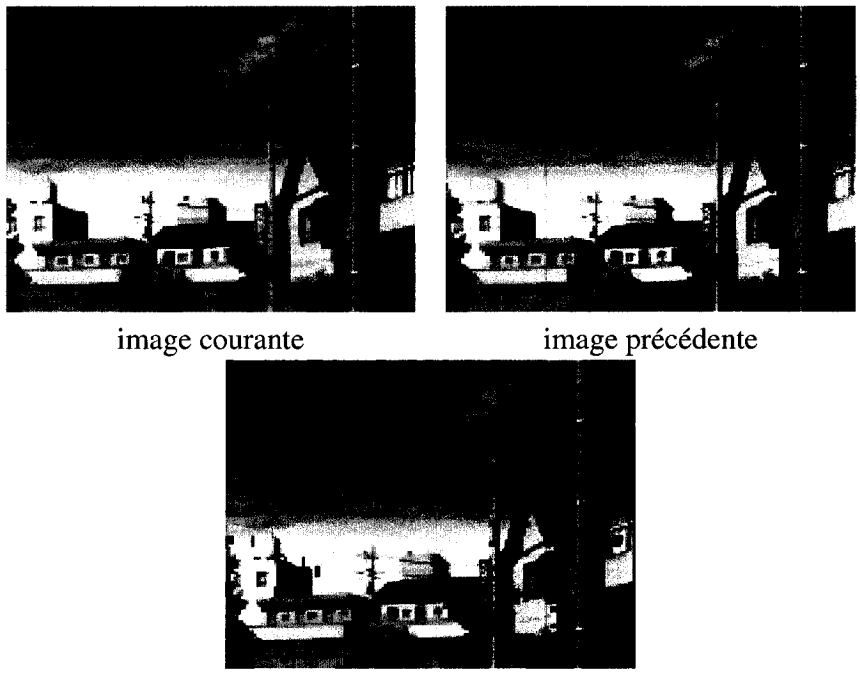


FIG. 2.3 Exemple de résultat obtenu avec l'algorithme simple

FIG. 2.4 Résultat de l'interpolation avec la version optimisée de l'algorithme simple

premier temps, on calcule l'image interpolée en considérant les vecteurs de mouvement des blocs issus de l'image courante. On calcule ensuite une deuxième image où les mouvements considérés sont ceux des blocs de l'image précédente. On fusionne ensuite les deux images en procédant de la façon suivante : l'image issue des mouvements des blocs de l'image courante sert de base. Les pixels laissés intacts sur cette image (en noir donc) sont remplacés par les pixels de la deuxième image. La figure 2.4 montre un exemple de résultat obtenu en utilisant un tel algorithme. Bien entendu, la qualité est meilleure puisque les rectangles sombres sont effacés mais un certain nombre d'artefacts restent visibles sur l'image.

En réalité, l'algorithme simple est davantage une base commune à tous les algorithmes de MC-FRC basés sur des techniques d'estimation de mouvements par blocs (BME). En effet, il faut d'abord estimer le mouvement des blocs. Une fois les vecteurs de déplacements obtenus, l'algorithme doit s'assurer, d'une part, de leur validité et, d'autre part, de la façon de les utiliser correctement afin d'obtenir un résultat convainquant. S'assurer de la validité d'un vecteur est une tâche complexe puisqu'elle requiert de posséder

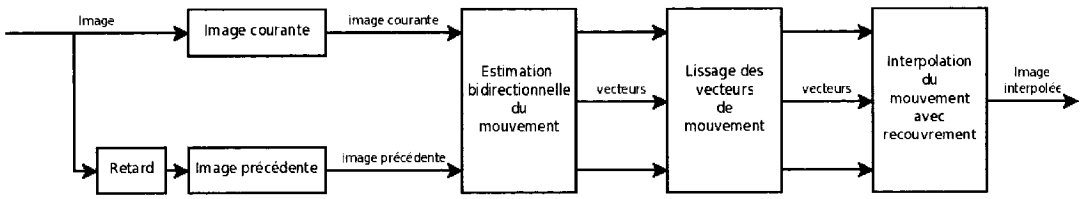


FIG. 2.5 Schéma de l'algorithme complexe

une métrique d'erreur. C'est cette difficulté qui différencie le fonctionnement d'un codec vidéo avec un algorithme de MC-FRC. Dans le cas des encodeurs vidéos, les erreurs éventuelles sur les vecteurs de mouvement sont corrigées par l'ajout d'une image représentant l'erreur entre l'image estimée par le mouvement et l'image réelle. Dans le cas de l'augmentation du débit, on ne possède pas l'image réelle et il est donc impossible de connaître l'erreur entre les deux images. Pour s'assurer de l'exactitude des vecteurs, on doit alors procéder à des raffinements ou bien introduire une mesure qui vise à détecter les vecteurs erronés. Ces deux solutions ajoutent en complexité et en temps à l'algorithme. L'utilisation des vecteurs est aussi une étape importante. Une utilisation simple (qui consiste à déplacer les blocs selon le vecteur estimé) présente beaucoup d'inconvénients dont le plus flagrant est la discontinuité observée dans l'image interpolée. Il faut donc opter pour des solutions plus évoluées comme le recouvrement des blocs. Tout ceci ajoute encore à la complexité globale de l'algorithme.

2.3 Algorithme complexe

L'algorithme complexe utilisé au cours des travaux présentés dans ce mémoire est basé sur l'algorithme présenté dans les travaux de Choi et al [9]. Cet algorithme utilise également une technique d'estimation de mouvement de bloc (BME) mais il cherche à affiner le résultat en appliquant un certain nombre de traitements supplémentaires.

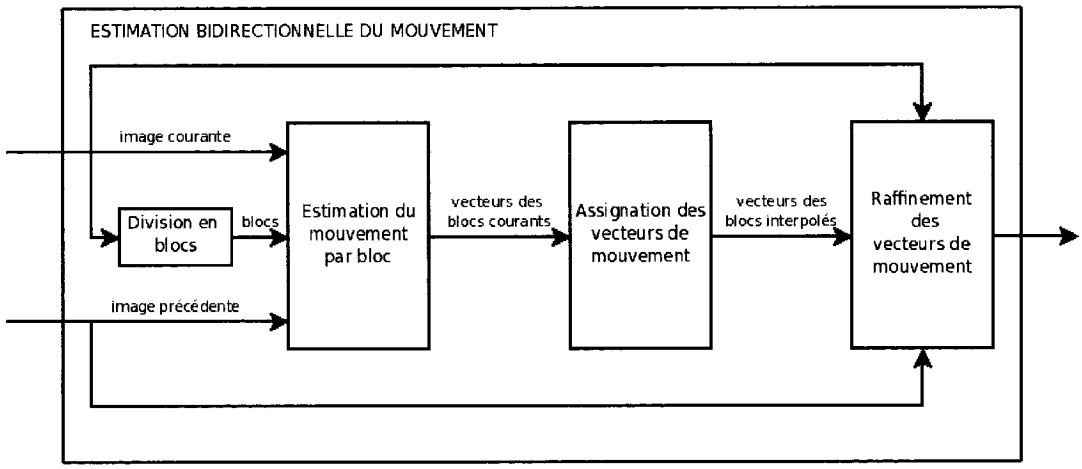
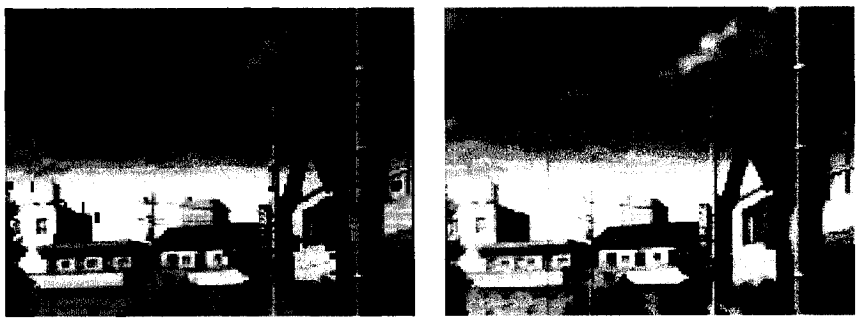


FIG. 2.6 Détails de l'estimation bidirectionnelle de mouvement

L'algorithme fonctionne en trois grandes étapes, comme montré dans la figure 2.5 : estimation de mouvements bidirectionnels, lissage des vecteurs de mouvement et interpolation par recouvrement de blocs. La première étape consiste donc à évaluer le mouvement des blocs. Toutefois, dans le cas de cet algorithme, on ne cherche plus à évaluer le mouvement des blocs de l'image courante ni de ceux de l'image précédente, mais bien de ceux de l'image qui sera interpolée. Pour cela, on crée une image vide qui deviendra l'image interpolée et on la découpe en blocs. On associe ensuite à chaque bloc un vecteur de mouvement qu'on a estimé en assignant des vecteurs issus de l'estimation du mouvement de l'image courante vers l'image précédente. L'assignation se fait en regardant quel bloc courant et le plus susceptible de correspondre au bloc de l'image interpolée considérée. Pour cela il suffit de regarder quel bloc de l'image courante, lorsqu'il est déplacé de la moitié de son vecteur de mouvement, recouvre le mieux le bloc de l'image interpolée. Le terme bidirectionnel provient donc du fait que les vecteurs trouvés permettent de déterminer la position du bloc de l'image interpolée dans l'image précédente ainsi que dans l'image courante. Le vecteur ainsi trouvé est ensuite raffiné pour offrir une meilleure qualité. Le raffinement consiste à faire tourner légèrement le



algorithme simple algorithme complexe

FIG. 2.7 Résultat de l'interpolation avec l'algorithme complexe comparé à l'algorithme simple

vecteur sur lui même et à conserver le meilleur vecteur. Ce dernier étant déterminé par un calcul de distance entre les blocs courants et précédents qui pointent à un endroit donné, le meilleur étant celui dont la distance est minimale. Cette partie de l'algorithme est détaillée dans le schéma de la figure 2.6. La seconde étape de l'algorithme, le lissage, consiste à comparer le vecteur de déplacement d'un bloc avec ceux de ses voisins afin d'éviter des erreurs de discontinuité. Enfin, la dernière étape consiste à créer l'image en utilisant ces vecteurs. La création de cette image utilise des techniques de recouvrement de blocs. On agrandit artificiellement la taille d'un bloc. Chaque pixel qui se retrouve dans une zone où plusieurs blocs se recouvrent est une moyenne des pixels issus de chacun des mouvements associés aux blocs. Ce procédé permet également d'éviter les discontinuités.

La figure 2.7 montre que les résultats de cet algorithme sont bien meilleurs que ceux de l'algorithme simple. Les images créées avec cet algorithme présente également un effet de flou qui provient de l'utilisation du recouvrement. Ce flou peut être aisément désactivé de façon logicielle, mais il permet également de corriger des défauts d'artéfacts trop visibles. Une fois intégrée à une vidéo, l'image apparaîtra sans défauts car le flou ne sera pas perçu étant, dans ce cas, un flou de mouvement classique, tandis que les

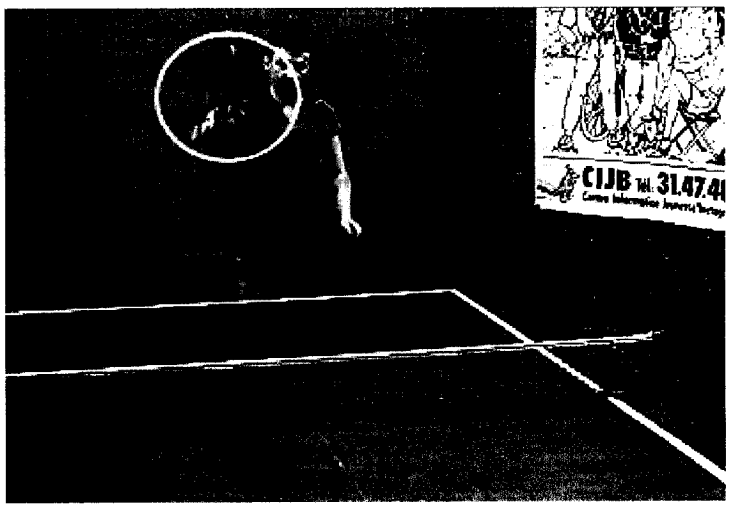


FIG. 2.8 Résultats avec défauts de l'algorithme complexe

défauts que le recouvrement permet de corriger pourraient être perçus. Toutefois, lorsque cet algorithme est confronté à des vidéos dans lesquelles les mouvements sont plus complexes, on observe toujours des défauts comme le montre la figure 2.8 avec notamment la raquette de tennis de table, ainsi que la main qui la tient, qui a disparu pour laisser la place à un ensemble informe de pixels.

En effet, bien que des efforts importants soient consentis pour améliorer la qualité de l'algorithme et de ses résultats, un certain nombre de défauts, inhérents à la technique utilisée, sont toujours visibles. Parmi les causes de ces défauts, on peut citer les objets invisibles (car beaucoup plus petits que la taille d'un bloc et dont l'influence sur les calculs de distance et donc sur la détermination des vecteurs de déplacement est très faible voire nulle), les collisions ou encore les mouvements trop amples. Le problème des objets invisibles est que la taille des blocs utilisée ne permet pas à l'algorithme de les repérer dans l'image et donc d'en tenir compte, ils risquent donc, entre autre, de disparaître de l'image interpolée. Une solution consiste à réduire la taille des blocs, mais elle est limitée par un aspect important de bruit. En effet, un bloc de petite taille est

plus susceptible d'être influencé par le bruit dans l'image. De plus, une petite taille ne permet pas de discriminer de manière efficace les blocs entre eux ; on risque donc de se retrouver avec plus d'erreurs d'estimation lorsqu'une petite taille est utilisée. Les collisions surviennent lorsqu'un objet est caché par un autre. Imaginons deux trains roulant en sens inverse, le train sur la voie la plus éloignée est caché par le second train lorsque ces derniers se croisent. Dans ce cas-ci, l'algorithme risque d'introduire des erreurs dans la zone de l'image où les deux trains se rejoignent car un des blocs disparait. C'est ce qu'on observe sur la figure 2.8 avec la raquette qui passe devant l'épaule du joueur. Le problème des mouvements trop amples quant à lui concerne davantage la taille de l'espace de recherche dans l'image. Ce dernier est le résultat de l'hypothèse que le mouvement dans l'image n'excède pas une valeur déterminée. Ainsi, pour que cela fonctionne, aucun objet ne doit se déplacer plus rapidement que la limite fixée. Bien entendu, une telle supposition est irréalisable dans une situation normale. Il suffit pour s'en convaincre d'imaginer un match de tennis au cours duquel les balles peuvent atteindre une vitesse de plus de 200 km/h qui peut se traduire par des vitesses à l'écran (mesurées en pixel par seconde) très importante selon la position de la caméra.

2.4 Problématique

Les deux algorithmes, le simple et le complexe, réalisent beaucoup de calculs de distance entre des blocs de pixels pour déterminer le mouvement dans l'image ainsi que pour améliorer cette estimation (dans le cas du complexe). Ils nécessitent donc un grand effort de calculs pour être réalisés. Dans le cas de l'algorithme simple, le coût des calculs peut être réduit en utilisant un algorithme de parcours de l'espace de recherche plus intelligent pour le BMA. Cependant, le nombre de calculs et la bande passante mémoire nécessaire restent élevés. Pour l'algorithme complexe, les différentes étapes effectuées requièrent davantage de calculs, surtout si l'on considère les divisions réalisées dans le

BMA.

C'est une grosse difficulté reliée à ces algorithmes. En effet cela empêche de les exécuter en temps réel sur un processeur RISC de faible complexité lorsque des flux vidéos de taille normale ou en Haute-Définition sont considérés. Le but étant de réaliser l'algorithme de MC-FRC en temps réel, on ne peut donc pas utiliser ce type de processeur. Les processeurs de grandes complexité sont généralement contre-indiqués pour une application embarquée, à cause de leur coût et de la puissance qu'ils consomment.

Pour se donner une idée de l'ampleur de la tâche à réaliser, considérons le cas de l'algorithme simple utilisant la méthode FS pour le BMA. Lorsque ce dernier est lancé sur un processeur Xtensa à architecture RISC (des chiffres très semblables seraient obtenus sur d'autres types de processeurs RISC 32 bits), il requiert 67 millions de cycles d'horloge pour interpoler une seule trame en travaillant sur des images de taille réduite (160 par 120 pixels). Si on considère une horloge de 300MHz pour ce processeur, il peut générer 4.4 images par seconde. C'est clairement insuffisant puisque par exemple passer de 50 Hz à 60 Hz impose de créer au moins 10 images par seconde.

La solution retenue pour ce projet est la création d'un jeu d'instructions spécialisé pour compléter le jeu standard. Le chapitre suivant présente les jeux d'instructions créés pour les deux algorithmes.

CHAPITRE 3

CRÉATION D'UN JEU D'INSTRUCTIONS SPÉCIALISÉ

Dans ce chapitre, les méthodes et solutions retenues lors de la création des jeux d'instructions spécialisés sont présentées. Ce chapitre présente également les jeux d'instructions créés et fait l'analyse de leur impact sur les algorithmes. Il débute par la section 3.1 qui présente le cas de l'algorithme simple. La section 3.2 présente ensuite le cas de l'algorithme complexe. Les sections 3.3 et 3.4 présentent respectivment les approches pour vérifier les instructions et pour les utiliser dans les programmes C.

Pour faciliter la compréhension de ce chapitre, il convient de présenter ici certaines des conventions adoptées pour décrire les jeux d'instructions. Pour se référer dans l'image, on utilise le pixel en haut à gauche comme base de départ. L'axe des abscisses est croissant de la gauche vers la droite, celui des ordonnées croit vers le bas. En d'autres termes, le pixel en haut à gauche est situé en (0,0) et celui en bas à droite en (w-1,h-1) où w et h sont respectivement la largeur et la hauteur de l'image. La position (x,y) d'un bloc dans une image est la position (x,y) de leur pixel en haut à gauche. Tous les blocs sont considérés carrés.

3.1 Le jeu d'instructions créé pour l'algorithme simple

Le jeu d'instructions créé pour l'algorithme simple et les résultats obtenus avec ce dernier ont fait l'objet d'une publication à la conférence SIPS2006 [3] ainsi qu'une soumission au Journal of DSP Technologies [4].

3.1.1 Registres pour la réutilisation : blocs et paramètres de l'algorithme

Pour l'algorithme simple, le jeu d'instructions a été créé en suivant un principe simple. Dans l'algorithme, on constate en effet que l'unité de base est le bloc d'une image. Les calculs sont faits entre les blocs et on s'intéresse au mouvement d'un bloc. Le jeu d'instructions spécialisé cherche donc à reproduire cet aspect de l'algorithme en intégrant ce module de base, le bloc, dans le processeur. Un bloc représente une surface carrée de pixels de l'image, ce sont donc des données sur lesquelles on réalise des calculs comme la somme des valeurs absolues des différences (SAD). Pour tirer parti de ce fait, des registres sont créés afin de contenir l'ensemble des pixels d'un bloc. Dans le cas du jeu d'instruction créé, ces registres sont larges de 128 bits et peuvent contenir des blocs de taille 4x4 pixels, où chaque pixel est représenté sur une valeur de 8 bits. Ce choix impose une limitation sur l'algorithme car la taille des blocs, originellement un paramètre de l'algorithme, est bloquée. Ces nouveaux registres sont créés dans un fichier de registre. Ceci permet une utilisation moins limitative que des registres d'état. Ces registres spécifiques seront appelés registres-bloc dans la suite de ce document.

Les registres d'états sont utilisés pour contenir les paramètres de l'algorithme ainsi que des données auxquelles les calculs font souvent appel. Notamment, les adresses mémoire des images traitées sont stockées dans des registres d'état spécifiques. Ces registres possèdent également l'avantage de permettre aux instructions de fonctionner avec plus de données que la limite architecturale (due à l'encodage des instructions) de trois valeurs.

3.1.2 Liste des instructions

Le tableau 3.1 présente une liste des instructions créées rangées par catégorie. Chaque instruction est décrite sommairement et le tableau indique également le nombre

Catégorie	Nom	Description	T	HK	Net	Cyc
Chargements	*cloadline*	Charge une ligne (4 pixels) du bloc de l'image courante selon la position (x,y) du premier pixel.	4	0	4	2
	ploadline	Idem mais pour l'image précédente.	4	0	4	2
Calculs	*dist*	Calcule la distance SAD entre un bloc courant et quatre blocs précédents.	192	0	192	2
Paramétriques	*indexc*	Calcule l'index d'un pixel dans le tableau à partir de sa position (x,y).	14	0	14	2
	wprecxy	Place dans un registre d'état la position du bloc précédent considéré.	2	2	0	1
	wcurxy	Idem mais pour le bloc courant.	6	6	0	1
	min_sas1	Calcule la taille de l'espace de recherche et ses limites en fonction de la position du bloc et des paramètres.	3	0	3	1
	min_sas2		3	0	3	1
	max_sas		3	0	3	1
	multi	Multiplieur 16 bits pour calculer y.w.	1	0	1	2
	movblr	Extrait les lignes d'un registre bloc et les place dans un registre normal	4	4	0	1
Automatiques	*ld.bloc128*	Instructions pour charger, sauvegarder et déplacer les registres bloc. Créées automatiquement par les outils pour que le compileur C puisse gérer le nouveau banc de registres.	1	0	1	1
	st.bloc128		1	0	1	1
	mv.bloc128		1	0	1	1

TAB. 3.1 Listes des instructions pour l'algorithme simple

d'opérations effectuées dans l'instruction ainsi que la nature de ces opérations. Les opérations équivalentes que réalise l'instruction peuvent en effet être de deux natures différentes : elles peuvent être nettes (colonne Net), c'est à dire qu'elles participent au déroulement de l'algorithme comme par exemple les opérations réalisées par l'instruction *dist*, ou elles sont présentes pour permettre le fonctionnement du jeu d'instructions et ne découlent pas de l'algorithme. Dans ce dernier cas, elles sont dites de maintenance (colonne HK pour House-Keeping) comme par exemple pour l'instruction *wcurxy*. Le total du nombre d'opérations effectuées par une instruction est indiqué dans la colonne T. Le nombre de cycles nécessaires pour réaliser l'instruction est fourni dans la colonne Cyc.

Quatorze instructions ont été décrites dans ce jeu d'instructions spécialisé. Elles sont créées pour travailler à partir des données contenues dans le nouveau fichier de registres. Elles réalisent donc les opérations de base qui sont effectuées avec les blocs par l'algorithme, à savoir les chargements et les calculs de distance SAD. Quelques instructions travaillent sur des parties répétitives de l'algorithme comme le calcul des limites de l'espace de recherche.

3.1.3 Description complète des instructions

3.1.3.1 Instructions de chargement

En premier lieu, il convient de remplir ces registres avec les données des pixels de l'image. Deux instructions, *st.Bloc128* et *ld.Bloc128*, sont créées automatiquement par les outils Tensilica. Elles réalisent des chargements et des sauvegardes de 128 bits de données situées à l'adresse indiquée en argument. Ces instructions sont créées pour permettre au compilateur C de pouvoir travailler avec le banc de registre de 128 bits. Elles ne sont toutefois pas utilisées par les programmes car elles ne correspondent pas

aux besoins de l'application. En effet les pixels d'un bloc ne sont pas situés à des emplacements continus en mémoire. Ces instructions ne chargent que des donnés continues en mémoire. Par conséquent, des instructions de chargement spécialisées ont été créées. Elles facilitent le chargement d'un bloc ainsi que la réutilisation des données. Ces instructions, *ploadline* et *cloadline*, sont utilisées pour charger une ligne de quatre pixels (32 bits) dans un registre-bloc. L'adresse est calculée à partir de la position (x,y) du premier pixel de la ligne, de la largeur de l'image w et du pointeur mémoire du tableau de pixels (contenu dans un registre d'état). Le décalage (offset) est calculé selon l'équation 3.1.

$$offset = x + w \cdot y \tag{3.1}$$

Pour les deux instructions, x et $w \cdot y$ sont passés en arguments pour éviter une multiplication à l'intérieur de l'instruction. De façon générale, l'algorithme évite les multiplications et, par conséquent, travaille autant que possible avec $w \cdot y$ plutôt que y. Cela ne change pas les propriétés de l'algorithme et cela facilite la gestion des tableaux de pixels. L'instruction *ploadline* est utilisée pour les pixels situés dans l'image précédente, *cloadline* pour ceux de l'image courante. La ligne chargée est placée en bas du registre-bloc, les autres lignes étant décalées vers le haut comme le montre la figure 3.1. Pour les blocs de 4 par 4 pixels, un total de 4 chargements doit donc être réalisé pour charger l'ensemble d'un bloc.

Un problème majeur est l'alignement obligatoire des chargements. En effet, l'architecture du processeur Xtensa ne supporte que les chargements de mots alignés. Ainsi, si l'on souhaite charger une ligne de 4 pixels, soit 32 bits, il faut qu'elle commence à une adresse multiple de 4. On peut faire des chargements de 64 ou 128 bits qui recouvriraient la valeur souhaitée, mais dans ce cas il faut également que le mot mémoire soit aligné ce qui signifie une adresse multiple de 8 ou 16 respectivement. Si l'on considère que le

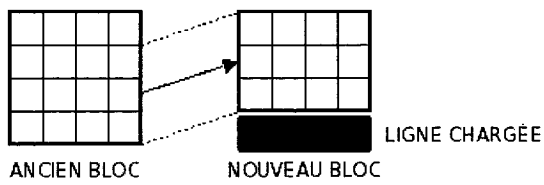


FIG. 3.1 Méthode de chargement d'une ligne dans un bloc

début de l'image, soit le pixel en haut à gauche, se trouve à l'adresse 0, on peut donc charger aisément dans les registres les blocs dont la position en x est un multiple de 4. Dans tous les autres cas, il faudra charger les deux blocs alignés qui recouvrent le bloc concerné. Donc, si l'on souhaite charger le bloc en x=3, il faut charger les deux blocs en x=0 et en x=4.

3.1.3.2 Instructions de calcul

Pour les calculs dans le contexte de l'algorithme simple, une seule instruction a été créée. Cela provient du fait que seuls les calculs de distance SAD sont réalisés sur les blocs dans les algorithmes de BMA. L'instruction *dist* est une implantation matérielle du calcul de la SAD. Elle calcule la distance entre les blocs courants et précédents comme montré sur la figure 3.2. En réalité, elle calcule quatre distances par appel en utilisant deux blocs précédents comme argument au lieu d'un. Comme ces blocs sont adjacents, les pixels qu'ils contiennent représentent 5 blocs (par recouvrement). L'instruction corrige donc le problème de l'alignement en augmentant ses capacités de calculs. L'instruction ne réalise que quatre calculs de distance au lieu des cinq possibles car le dernier bloc sera calculé lors des calculs des blocs voisins. La figure 3.3 montre comment l'instruction *dist* calcule les 4 distances en parallèle. L'instruction calcule et conserve aussi le minimum et sa position. Le nombre d'opérations qui sont réalisées à l'intérieur de cette instruction se comptabilise ainsi : pour un bloc de 4 par 4 pixels, la distance SAD est

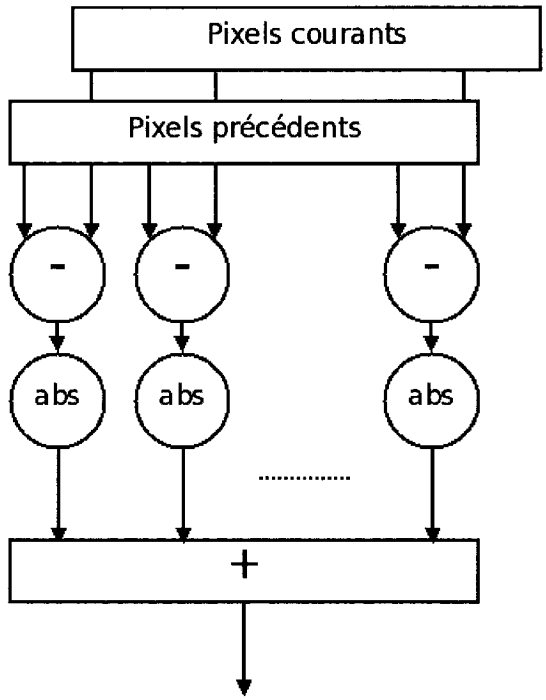


FIG. 3.2 Calcul de la distance SAD

constistué de 16 soustractions suivies de 16 calculs de valeur absolue puis de 15 additions, soit un total de 47 opérations par bloc. L'instruction calcule 4 distances SAD par appel et réalise également un certain nombre d'opérations supplémentaires pour calculer le minimum. Au total donc, 192 opérations sont réalisées dans l'instruction à chaque appel (voir tableau 3.1). Cette instruction est aussi par conséquent la plus complexe et celle qui requiert le plus de matériel du jeu d'instructions.

3.1.3.3 Autres instructions

Le reste des instructions gère les paramètres de l'algorithme ainsi que des composants mineurs.

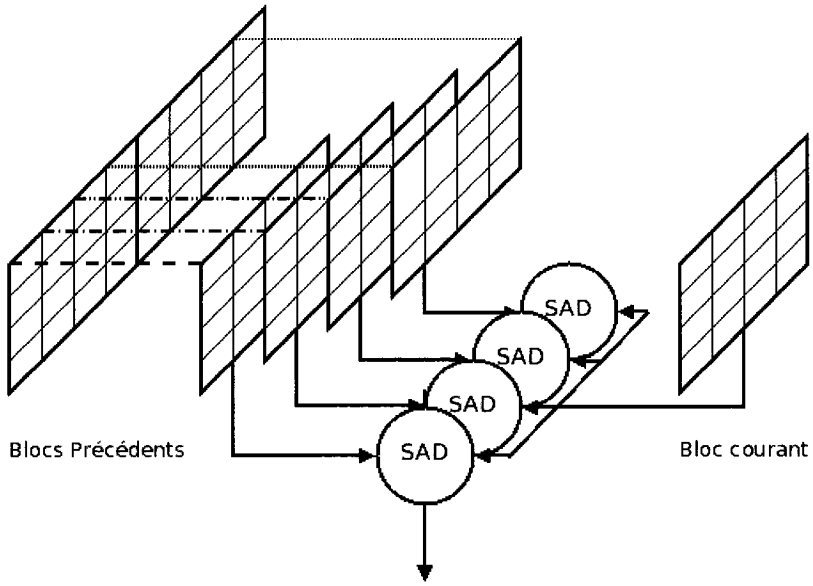


FIG. 3.3 Fonctionnement de l'instruction *dist*

Pour les paramètres, quatre instructions ont été créées : *indexc*, *wprecxy*, *wcurxy* et *multi*. La première calcule le décalage comme montré dans l'équation 3.1. Les deux instructions suivantes sont utilisées pour charger les valeurs des divers paramètres dans les registres d'état et permettre ainsi le bon fonctionnement des instructions de chargement et de calculs qui travaillent avec ces registres. Ces deux instructions sont donc des instructions de gestion du jeu étendu. Enfin, la dernière instruction est utilisée pour réaliser certaines multiplications entre des paramètres de l'algorithme.

Les instructions *max_sas*, *min_sas1* et *min_sas2* sont, comme indiqué dans le tableau 3.1, chargées de déterminer les limites de l'espace de recherche. Ces limites sont calculées à partir de la position du bloc courant considéré et de différents paramètres comme la taille de l'image et la taille de l'espace de recherche.

L'instruction *movblr* est utilisée pour extraire les lignes d'un bloc. On lui passe en argument le bloc auquel on est intéressé ainsi que le numéro de la ligne du bloc

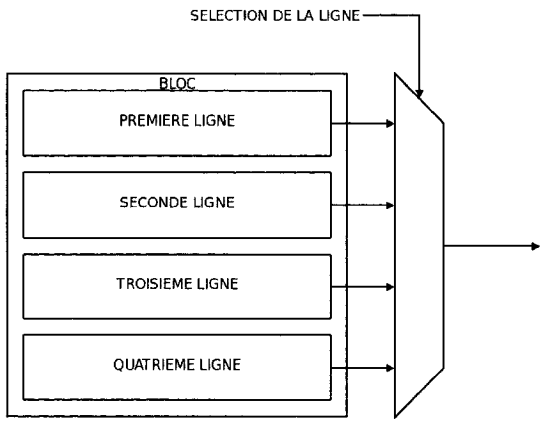


FIG. 3.4 Fonctionnement de l'instruction *movblr*

qu'on veut extraire comme le montre la figure 3.4. Cette instruction permet d'extraire le contenu d'un bloc ce qui peut notamment faciliter le déverminage mais est surtout utile lorsque l'on veut écrire en mémoire le contenu du bloc. En effet, les lignes du bloc ne sont pas contigues en mémoire, il faut donc les extraire une à une pour aller ensuite les placer au bon endroit.

3.2 Le jeu d'instructions créé pour l'algorithme complexe

3.2.1 Liste des instructions

Le tableau 3.2 présente l'ensemble des instructions créées pour l'algorithme complexe. Comme pour les instructions de l'algorithme simple, elles sont rangées par catégorie. On présente également le nombre d'opérations effectuées dans ces instructions ainsi que leur nature.

TAB. 3.2 Liste des instructions pour l'algorithme complexe

Catégorie	Nom	Description	T	HK	Net	Cyc
Chargements	*cloadline*	Charge une ligne de 8 pixels de l'image courante selon la position (x,y) de son premier pixel.	6	0	6	2
	cloadline2	Charge la ligne suivante (x+8,y).	6	0	6	2
	ploadline	Idem mais pour l'image précédente.	6	0	6	2
	ploadline2		6	0	6	2
Calculs	*sad_line*	Calcule la distance SAD entre une ligne du bloc courant et les lignes de huit blocs précédents.	184	0	184	2
	divd_64	Division euclidienne sur les résultats intermédiares de *sad_line.*	4	0	4	1
	div_64	Instruction de division euclidienne.	3	0	3	1
	adiv2	Ajoutte la moitié du second argument au premier.	2	0	2	1
Limites	*start_end_y*	Calculent à partir de la position du bloc quels pixels et quelles lignes sont hors de l'image.	16	0	16	2
	calcmask		19	0	19	1
Paramétriques	*addw*	Addition spécifique aux paramètres retenus dans des registres d'états.	3	3	0	1
	wrs	Initialise des registres spécifiques.	2	2	0	1
	bpos	Détermine la position d'un pixel à l'intérieur d'un bloc (centre ou frontières) pour la procédure de recouvrement.	5	0	5	1
	bound	Détermine si le mouvement fait sortir le pixel de l'image.	19	0	19	1
	indexc	Calcule l'adresse mémoire du pixel de l'image courante à partir de sa position (x,y).	5	0	5	1
	indexp	Idem pour l'image précédente.	5	0	5	1
	dmin	Surveille et conserve le minimum et sa position pour les calculs de la distance SAD.	3	3	0	1

3.2.2 Approche retenue pour l'optimisation de la réutilisation des données

Pour cet algorithme, il n'était pas possible d'utiliser le principe des registres-bloc. En effet, la taille du bloc est une composante importante de la qualité de l'algorithme. Une taille de quatre est trop petite et est susceptible de provoquer des erreurs de prédiction, notamment en raison de la sensibilité au bruit et de la faible capacité discriminante entre les différentes distances SAD. Par conséquent, l'algorithme complexe, qui a pour objectif une grande qualité d'image, utilise une taille de bloc de 8 par 8. Cette taille ne tient pas dans un registre standard (un registre de 512 bits serait nécessaire pour contenir les 64 pixels requis). L'idée retenue ici est donc de travailler avec les lignes des blocs plutôt qu'avec l'intégralité du bloc. Il faut donc travailler avec les 8 lignes du bloc successivement pour faire l'intégralité d'un bloc.

Les instructions créées pour charger les données des pixels sont *ploadline*, *cloadline*, *ploadline2* et *cloadline2*. Elles chargent des mots de 64 bits, soit une ligne de 8 pixels, depuis la mémoire en calculant l'adresse à partir de la position (x,y) du premier pixel de la ligne. Les lignes sont stockées dans des registres de 128 bits qui peuvent donc contenir deux lignes. L'utilisation de registres deux fois plus longs qu'une simple ligne de 8 pixels est imposée par les restrictions sur les chargements qui se doivent d'être alignés comme cela a déjà été évoqué. En chargeant deux lignes alignées consécutives, on peut travailler sur toutes les lignes non alignées recouvertes par les deux lignes. Deux de ces registres sont créés, l'un pour les pixels issus de l'image courante et l'autre pour ceux de l'image précédente. Les instructions *cloadline* et *cloadline2* travaillent avec les pixels de l'image courante tandis que *ploadline* et *ploadline2* s'occupent des pixels de l'image précédente.

Comme les nouvelles instructions travaillent avec les lignes plutôt qu'avec l'intégralité du bloc comme cela avait été fait pour le jeu d'instruction de l'algorithme simple, il faut maintenant être capable de retenir les valeurs intermédiaires du calcul

de la distance SAD. En effet, il faut accumuler les distances calculées sur les lignes pour obtenir la distance sur le bloc. Un registre a donc été créé pour contenir les valeurs intermédiaires. Les calculs se font donc en utilisant ce registre intermédiaire, les distances calculées sont ajoutées à la valeur de ce registre. L'instruction qui effectue les calculs de distance SAD, *sad_line*, travaille avec les registres contenant les pixels des lignes. Comme ces registres permettent d'accéder à 8 lignes de 8 pixels chacune, on peut donc réaliser un total de 8 calculs de distance entre la ligne du bloc courant et les lignes des 8 blocs précédents stockées dans le registre comme le montre la figure 3.5. Le calcul de la SAD se faisant de la même façon que pour l'algorithme simple, c'est à dire comme cela est montré dans la figure 3.2. Le registre contenant les valeurs intermédiaires contient donc 8 valeurs de distance de 32 bits chacune soit un registre de 256 bits (ce registre pourrait certainement être réduit en taille sans nuire à sa fonctionnalité). Dans certains cas, ce grand nombre de calculs ne sera pas utile car une seule distance sera nécessaire mais cette méthode permet de faire levier sur les données chargées et peut fournir une grande accélération dans certain cas. Quand cela survient, l'instruction réalise 8 soustractions suivies de 8 calculs de valeur absolue et de 7 additions pour chaque ligne de 8 pixels pour un total de quelque 180 opérations pour l'instruction au complet.

3.2.3 Le problème des frontières de l'image

Dans le cas de l'algorithme complexe, les frontières de l'image posent problème. En effet, lorsque le bloc considéré pour le calcul de la distance SAD est partiellement en dehors de l'image, il faut modifier le calcul pour ne pas introduire d'erreurs. On ne peut pas en effet compléter les pixels manquants (situés hors de l'image) par une valeur nulle ou fixe, car cela introduirait des erreurs de calculs de distance. La solution retenue est d'utiliser des masques qui valident ou invalident un pixel dans une ligne. Cela permet de savoir si ce pixel doit être considéré dans le calcul de la distance. L'instruction *calcmask* calcule le masque qu'il faut appliquer aux lignes du bloc en fonction de sa position en

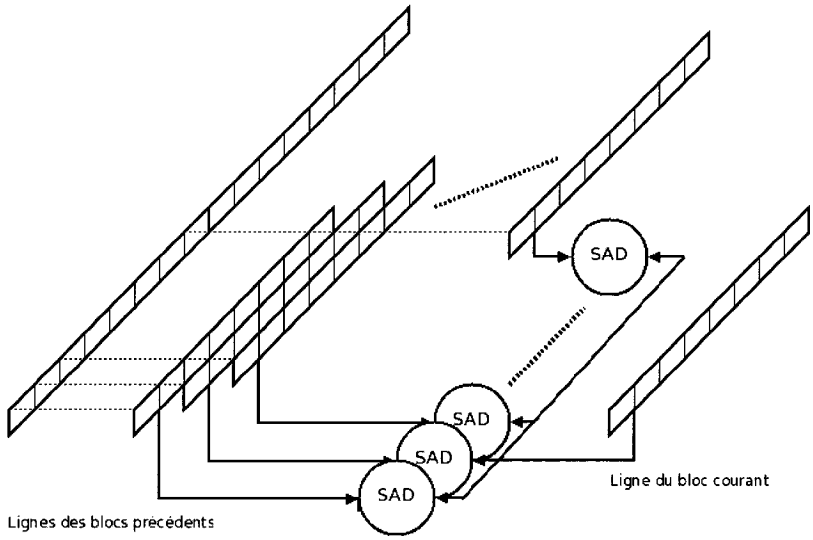


FIG. 3.5 Distance SAD sur les 8 lignes

x. Deux masques sont calculés à chaque calcul de SAD, celui du bloc courant et celui du bloc précédent. Chaque bit du masque correspond à un pixel. Une autre instruction, *start_end_y*, calcule quelles lignes du bloc sont valides en fonction de la position en y de ce dernier et doivent être incluses dans le calcul de la SAD.

Une autre conséquence est qu'il faut pouvoir comparer les différentes distances obtenues. Un bloc situé partiellement en dehors de l'image est susceptible d'avoir une distance plus faible. L'algorithme complexe utilise donc une distance normalisée sur le nombre de pixels, à savoir la moyenne de la SAD (Mean Absolute Difference - MAD). Cela implique des calculs de division au sein de l'algorithme. Deux instructions, *div_64* et *divd_64*, se chargent d'effectuer les divisions à l'aide d'une table de valeurs contenant les inverses des nombres situés de 0 à 64 inclus (pour éviter tout problème de division par zéros, le cas zéros retourne 0) et multiplié par 2^{15}. Les instructions multiplient la valeur divisée par la valeur trouvée dans cette table à la position du diviseur et décalent de 15 bits vers la droite le résultat trouvé. *Divd_64* travaille directement avec les valeurs

intermédiaires stockées dans le registre de 256 bits.

3.2.4 Machine à états, contrôle et autres instructions

Les calculs et les chargements mémoire s'accélèrent aisément. L'idée de base est d'augmenter les capacités de calculs et de mémoire, via de nouveaux registres, du processeur ainsi que la bande passante mémoire. Cependant, quand l'algorithme est dominé par le contrôle, la tâche est plus complexe. Un arbre de décision peut difficilement être résumé en une simple instruction et ce d'autant plus si le nombre de facteurs entrant en compte dans la décision est important, ceci à cause de la limitation du nombre d'arguments de l'instruction. Pour l'algorithme complexe, deux instructions ont été créées pour accélérer les décisions au sein de l'algorithme. Ces instructions peuvent être vues comme calculant un état à partir d'un certain nombre de paramètres intégrés aux registres du processeur. *Bpos* calcule la situation d'un pixel d'un bloc afin de savoir si le pixel est situé sur le bord du bloc et est donc un candidat pour le recouvrement. *Bound* détermine si le déplacement effectué fait sortir le pixel de l'image. L'impact sur la performance de ces instructions est relativement peu élevé comparé aux instructions de calcul car elles ne réalisent pas un grand nombre d'opérations. Par exemple, bound réalise 2 soustractions, 2 additions, 8 comparaisons et 7 ET logiques soit un total de 19 opérations.

Le reste des instructions sert pour gérer les paramètres et supporter un certain nombre d'éléments mineurs de l'algorithme. Cinq instructions sont concernées : *indexc*, *indexp*, *addw*, *wrs* et *adiv2*. Les deux premières calculent l'index dans le tableau des pixels, de l'image courante et de l'image précédente respectivement, à partir de la position et du mouvement considérés. *wrs* et *addw* travaillent sur les registres contenant les paramètres. La première instruction permet de les initialiser selon les besoins et la seconde réalise des calculs permettant leur mise à jour à mesure que l'algorithme progresse dans son fonctionnement. Enfin, *adiv2* réalise l'addition de deux arguments en divisant

en premier lieu le deuxième argument par 2. Cette instruction est utilisé pour ajouter le vecteur de déplacement divisé par 2 à la position du pixel considéré afin d'obtenir sa position une fois interpolé.

3.3 Vérification fonctionnelle des instructions spécialisées et déverminage

La création d'un jeu d'instructions est nécessairement sujette à erreurs. Il faut donc s'assurer que les instructions créées vérifient les spécifications fonctionnelles. Ce processus est similaire au processus de vérification qu'on applique dans la création de puces spécialisées. En effet, le jeu d'instructions est créé dans un langage HDL similaire au langage Verilog. Les erreurs qu'on peut observer sont nombreuses comme, par exemple, des erreurs de branchements ou de troncature qui faussent l'exactitude des calculs. Toutefois, la vérification ne se fait pas à l'aide de bancs d'essai qui introduisent des valeurs selon un algorithme de test. Ici, le banc d'essai est un programme C qui utilise l'instruction et s'assure que les résultats qu'elle retourne correspondent aux résultats attendus. Le programme d'essai est donc auto-vérifiant. Les tests peuvent être réalisés de façon exhaustive ou semi-aléatoire. Lorsqu'un problème est rencontré, le concepteur doit analyser les causes de ce problème et le résoudre en corrigeant l'instruction. Par exemple, l'instruction qui réalise une division, *div_64*, peut simplement être vérifiée de façon exhaustive à l'aide d'une boucle qui divise un nombre donné par toutes les valeurs allant de 1 à 64. La figure 3.6 montre un exemple de ce type de banc d'essai auto vérifiant.

Un autre problème survient lorsque les instructions sont utilisées dans un programme complexe. Comme les instructions peuvent être complexes, leur utilisation peut entraîner, si elle est faite de façon désordonnée, des bogues dans le programme. Il faut donc vérifier également que le programme créé utilisant le jeu étendu est correct. Pour ce faire, il faut comparer les résultats d'un programme référence dont le fonctionnement

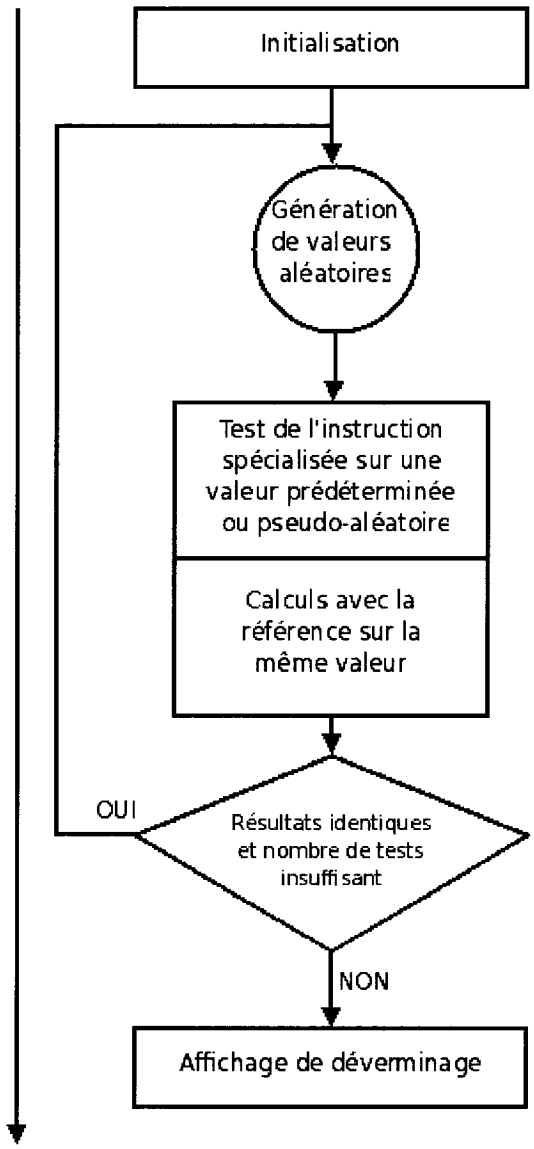


FIG. 3.6 Programme d'essai d'une instruction autovérifiant

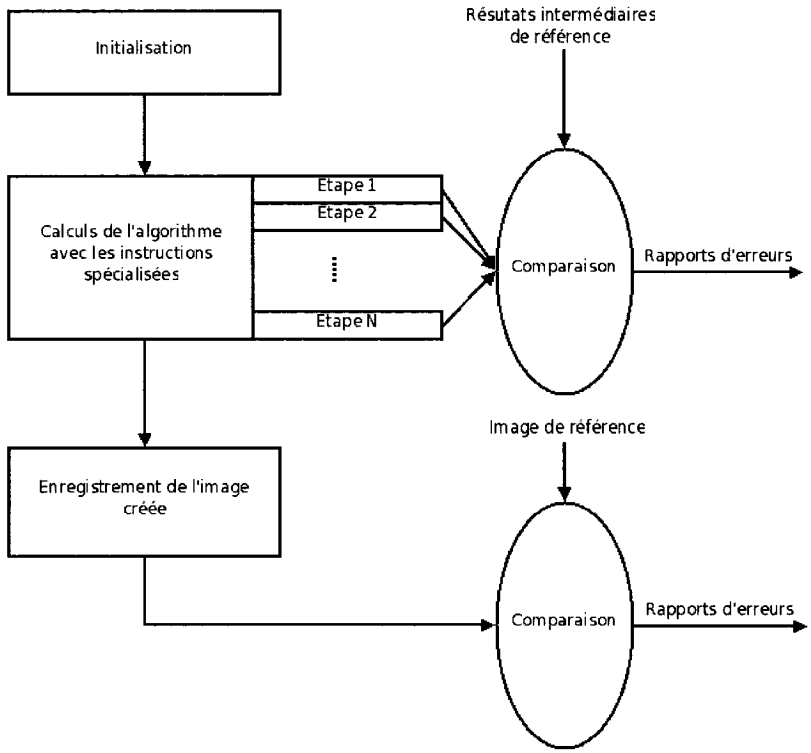


FIG. 3.7 Vérification fonctionnelle d'un programme utilisant des instructions spécialisées

est sûr. Bien entendu, comparer le résultat final risque de ne pas permettre un déverminage aisé. Il faut donc définir des points de comparaison qui serviront à vérifier les étapes de l'algorithme. Par exemple, dans le cas de l'algorithme complexe, les vecteurs de mouvement estimés sont utilisés comme valeurs de comparaison et les points de référence sont situés après les étapes d'estimation bi-directionnelle et de lissage. Dans le cas de l'algorithme simple, les vecteurs sont également utilisés comme éléments de comparaison pour s'assurer du fonctionnement correct des programmes utilisant le jeu étendu. Un exemple de vérification fonctionnelle par étapes d'un programme utilisant des instructions spécialisées est montré dans la figure 3.7.

3.4 Modifications apportées aux programmes C

3.4.1 Programmes C créés

Une fois les nouveaux jeux d'instructions créés, il faut modifier les programmes C pour que ces derniers puissent les utiliser. La modification des programmes laisse un grand degré de liberté qui peut permettre d'atteindre une large gamme de performance. Cependant, ce degré de liberté est conditionné par le jeu d'instructions créé qui peut s'avérer extrêmement restrictif. Comme on va le voir, le jeu d'instructions de l'algorithme simple est plus libre que celui de l'algorithme complexe.

Pour l'algorithme simple, un total de six versions a été programmé. Quatre versions utilisent l'algorithme Full Search (FS) comme algorithme de BMA et deux l'algorithme One-Dimensional Full Search (ODFS). Pour ceux utilisant FS, trois versions (nommées FS-TIE1, FS-TIE2 et FS-TIE3) utilisent le jeu d'instructions étendu et une version (appelée FS) sert de base de référence. Les différences entre les versions se situent dans la manière dont elles utilisent les nouvelles instructions pour parcourir l'espace de recherche. Pour ODFS, il existe une version avec les instructions spécialisées (ODFS-TIE) et une sans (ODFS). Les programmes basés sur le nouveau jeu d'instruction commencent tous par charger le bloc courant de 4 par 4 pixels dans un registre-bloc. Ensuite ils réalisent les calculs de distance avec les blocs précédents qui sont chargés au fur et à mesure.

FS-TIE1 travaille avec deux blocs de données de l'image précédente et traverse l'espace de recherche ligne par ligne. Le programme décale donc les données du plus récent bloc dans le registre contenant les données du plus ancien et recharge le registre ainsi libéré avec les données du prochain bloc. Donc, pour chaque appel à l'instruction dist, qui réalise les calculs, il faut déplacer un bloc puis charger un autre bloc au complet. FS-TIE2 travaille aussi avec deux blocs précédents mais parcours l'espace de

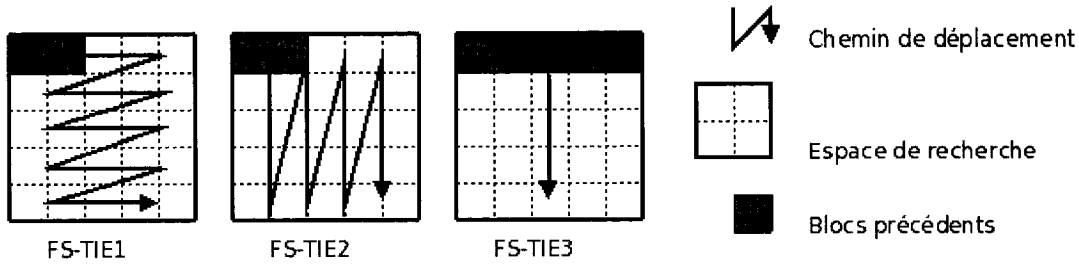


FIG. 3.8 Parcours de l'espace de recherche par le programmes FS-TIE

recherche colonnes par colonnes. Il lui suffit donc de charger deux nouvelles lignes, une pour chaque bloc, pour pouvoir faire les calculs suivants. Grâce à l'instruction ploadline qui se base sur ce principe, cette opération est facilitée. Ce programme augmente donc la réutilisation des données par rapport à FS-TIE1. FS-TIE3 est basé sur un principe similaire mais procède en une seule colonne qui prend la largeur de l'espace de recherche. Cela augmente de façon significative la réutilisation. La figure 3.8 montre comment ces trois programmes parcourent l'espace de recherche.

Les restrictions architecturales qui imposent des chargements alignés impliquent des changements sur l'algorithme ODFS lorsque ce dernier utilise le jeu d'instructions étendu (programme ODFS-TIE). En effet, lorsque l'algorithme travaille sur une colonne non alignée, il rencontre à nouveau la difficulté qui découle de l'obligation d'effectuer des chargements alignés. La solution retenue est de charger les deux colonnes recouvrant la colonne non alignée et de faire les calculs sur ces deux colonnes. La conséquence est que le programme ODFS-TIE réalise plus de calculs de distance que la référence ODFS. Il est donc susceptible de trouver des résultats différents (néanmoins meilleurs car c'est toujours le minimum qui est retenu). La figure 3.9 montre comment les deux programmes opèrent lorsqu'ils travaillent sur une colonne.

Le grand nombre de programmes créés à partir des instructions spécialisées de

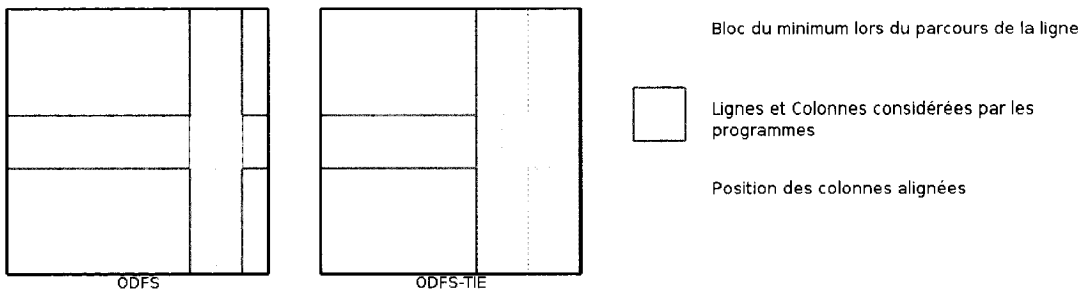


FIG. 3.9 différence entre ODFS et ODFS-TIE

l'algorithme simple montre que ces dernières sont très génériques et peuvent s'adapter à de nombreuses situations même si c'est parfois au prix d'une modification de l'algorithme.

Pour ce qui est de l'algorithme complexe, seuls deux programmes ont été créés. La version de référence qui n'utilise pas le jeu d'instructions étendu et la version accélérée qui se base sur ce dernier. Dans cette dernière version, de nombreuses modifications ont été réalisées pour tirer au mieux parti des instructions spécialisées et atteindre un facteur d'accélération important. Ainsi, les calculs de distance SAD, qui font appel à l'instruction sad_line, sont optimisés en fonction des situations dans lesquelles ils sont réalisés (raffinement, determination du vecteur ...).

3.4.2 Exemple de modifications

Pour les deux algorithmes, lorsqu'ils travaillent avec les instructions spécialisées, les fonctions C sont intégralement réécrites pour finalement n'intégrer que des instructions spécifiques et du code de contrôle (boucle, conditions, etc.). Cela conduit à un code plus compact qui réduit la demande sur la cache de données. Les exemples de code suivants montrent le calcul de la distance SAD dans l'algorithme de référence ainsi qu'en

utilisant les jeux d'instructions spécialisés créés. Le code C de la référence est le suivant :

```
for(k=0;k<blocksize;k++)
{
  for(l=0;l<blocksize*current.w;l=l+current.w)
  {
    d=d+abs(current.pixels[x1+k+l+y1]-previous.pixels[x2+k+y2+l]);
  }
}
```

Pour l'algorithme simple utilisant son jeu d'instructions :

```
DIST(bc,~bpg,~bpd);
```

Pour l'algorithme complexe utilisant son jeu d'instructions :

```
CALCMASK(x1,x2);
for(k=START_END_Y(y1,y2);k<RUR_END();ADDW(k))
{
  CLOADLINE2(x1);
  CLOADLINE(x1);
  PLOADLINE(x2);
  PLOADLINE2(x2);
  SAD_LINE();
}
```

Comme on le voit, les instructions spécialisées réduisent considérablement la complexité du code. Cela n'est pas surprenant étant donné que cette complexité est intégrée aux instructions. Toutefois, elles obligent également le concepteur à repenser intégralement son code puisqu'il n'y a souvent plus rien de similaire entre le code référence et les codes utilisant des instructions spécialisées. De plus, pour bien utiliser les instructions spécialisées, le concepteur est obligé de coder à très bas niveau. Les fonctions présentées en majuscules dans les codes précédents sont en réalité des appels directs aux instructions portant le même nom. C'est donc un travail plus délicat que la simple programmation en C pour laquelle une grosse partie de l'optimisation peut être efficacement prise en charge par le compilateur. Avec les instructions spécialisées, cela n'est plus possible car le compilateur avec lequel nous avons travaillé ne dispose pas d'information lui

permettant de déduire comment ces instructions s'utilisent ainsi que les opérations que ces instructions réalisent. Des outils automatisés pourraient être utilisés pour créer ou tirer profit d'instructions déjà créées mais ils sont loin d'avoir la performance requise.

CHAPITRE 4

RÉSULTATS DE PERFORMANCE

4.1 Pour l'algorithme simple

Ici, on présente les résultats obtenus avec l'algorithme simple, notamment la performance des instructions spécialisées et les diverses études sur l'espace des paramètres.

4.1.1 Superficie supplémentaire et fréquence d'horloge maximum du nouveau jeu d'instructions

Les instructions spécialisées créées pour accélérer l'algorithme simple ajoutent un total de 50515 portes aux 64348 portes du processeur de base. La superficie additionnelle est donc d'environ 79%. Ce résultat serait plus faible si l'on tenait compte de la superficie des mémoires de la cache. De plus, la complexité matérielle des instructions supplémentaires pourrait être réduite aisément si la taille du banc de registres utilisé était réduite à 8 au lieu de 16 sans impact pour ce qui est des cas présentés dans ces travaux. Toutefois, cette réduction risquerait de pénaliser certains cas pour lesquels 8 registres sont insuffisants (programme FS-TIE3 dans le cas d'un grand espace de recherche). La fréquence d'horloge maximale autorisée par le nouveau jeu d'instruction est de 271 MHz ce qui est supérieur à celle du processeur de base. La fréquence d'horloge du processeur n'est donc pas modifiée.

TAB. 4.1 Accélération de l'algorithme simple : cas FS

	FS	FS-TIE1	FS-TIE2	FS-TIE3
cycles	56040	1250	618	480
accélération relative	1	44.83	90.68	116.75
		1	2.02	2.6

TAB. 4.2 Accélération de l'algorithme simple : cas ODFS

	ODFS	ODFS-TIE
cycles	11360	467
Accélération	1	24.33

4.1.2 Accélération : FS et ODFS

Les accélérations obtenues pour l'algorithme simple sont présentées dans les tableaux 4.1 (FS) et 4.2 (ODFS). Elles s'étalent de 44.83 jusqu'à 116.75 pour l'algorithme simple utilisant FS et atteignent 24.33 pour l'algorithme simple avec ODFS, avec toutefois la remarque que dans ce dernier cas le nombre de calculs est plus important dans ODFS-TIE. La première ligne des deux tableaux montre le nombre de cycles requis pour réaliser l'interpolation d'un bloc de l'image courante. La seconde ligne montre l'accélération par rapport à la référence. La dernière ligne du tableau 4.1 montre que les instructions offrent une grande liberté qui peut conduire à des gains de performance de 160% lorsque elles sont utilisées de façon plus efficace. Les résultats sont obtenus avec une image de 160 par 120 pixels, pour une taille de bloc de 4 par 4 et un espace de recherche s'étendant de 8 pixels dans toutes les directions à partir de la position du bloc courant.

TAB. 4.3 Impact de la politique et de la taille de la cache sur la performance

Associativité	Taille (octets)	FS	FS-TIE2	Accélération
1	1024	62650	688	91.06
	2048	60840	662	91.9
	4096	59910	635	94.35
2	1024	56970	675	84.4
	2048	56040	618	90.68
	4096	55890	604	92.53
4	1024	55880	630	88.7
	2048	55800	616	90.58
	4096	55790	600	92.98

4.1.3 Impact de la taille de la cache et des délais mémoire

Ici, on s'intéresse à l'impact des paramètres mémoire sur les performances du jeu d'instructions étendu. Dans un premier temps, on regarde le rôle de la cache de données en modifiant sa configuration, i.e. sa politique et sa taille. Ceci est fait grâce au simulateur qui permet de modifier la cache simulée sans recréer un processeur. Le programme considéré est FS-TIE2 que l'on compare à la référence FS. Les paramètres de l'algorithme n'ont pas changé (même taille d'image, de bloc et d'espace de recherche). Les résultats sont consignés dans le tableau 4.3.

On regarde maintenant les performances lorsque l'accès mémoire en lecture est ralenti par une mémoire externe avec une grande latence. La cache est configurée selon les paramètres d'origine. Le delai standard de la mémoire est considéré comme nul, ce qui signifie qu'elle répond instantannément aux requêtes qu'elle reçoit. On augmente son delai jusqu'à 100 cycles de latence. Les résultats sont présentés dans le tableau 4.4.

TAB. 4.4 Impact de la latence de la mémoire en lecture sur la performance

Latence mémoire en lecture	FS	FS-TIE2	Accélération
0	56040	618	90.68
10	56510	678	83.35
20	56980	737	77.31
50	58380	920	63.46
100	60720	1220	49.77

4.1.4 Impact des paramètres de l'algorithme

Dans cette section, on s'intéresse aux paramètres de l'algorithme et à leur impact sur les performances. L'algorithme simple possède trois paramètres. La taille du bloc, la taille de l'espace de recherche et la taille de l'image. Comme l'utilisation du jeu étendu fixe la valeur de la taille des blocs à 4, seuls les deux derniers paramètres sont analysés. On considère ici le programme FS-TIE2. Les résultats sont retranscrits dans la figure 4.1. On augmente l'espace de recherche de 4 à 24 par pas de 4 afin de voir comment la taille de ce dernier modifie la performance. Un plus grand espace de recherche améliore la prédiction du mouvement car il permet de chercher dans une plus grande section de l'image. Ainsi, si les mouvements sont amples ils seront mieux déterminés avec un grand espace de recherche. La taille de l'image considérée est 160 par 120 pixels. Ici, le nombre de cycles considéré est celui qu'il faut pour traiter toute l'image soit 1200 blocs. Le nombre de cycle par bloc peut donc être retrouvé en divisant ce résultat par 1200.

Le tableau 4.5 présente l'impact de la taille de l'image sur les performances de l'algorithme. Dans ce cas, la taille de l'espace de recherche utilisé est de 8. Il est important d'obtenir une indication sur les performances de l'algorithme quand des images larges sont considérées car les formats NTSC ou la télévision HD utilisent des tailles supérieures à 640 par 480 pixels.

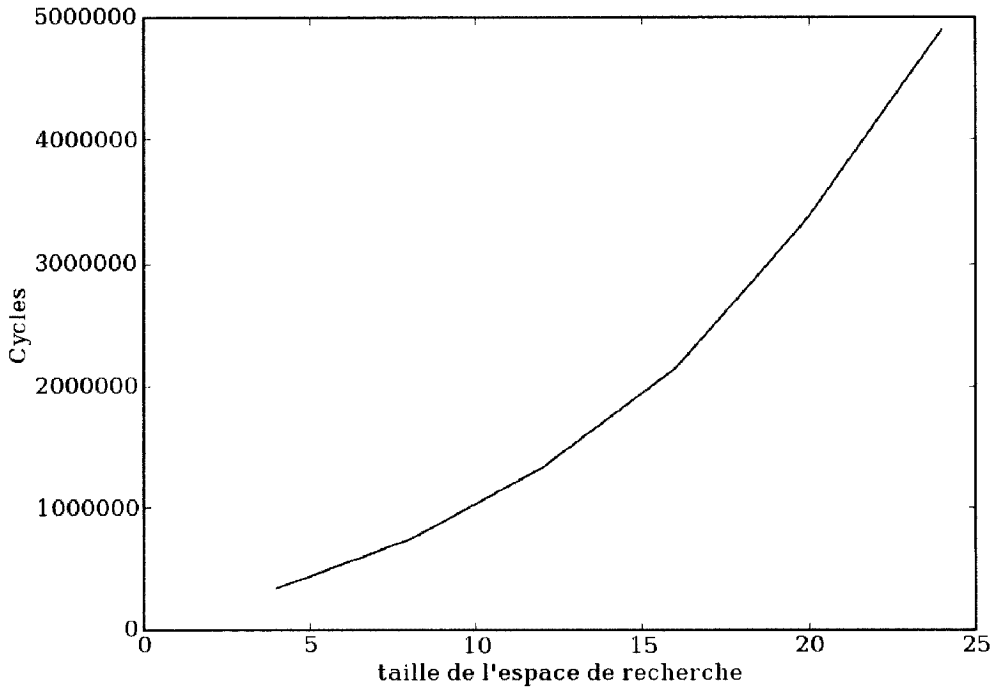


FIG. 4.1 Impact de l'espace de recherche sur la performance

TAB. 4.5 Impact de la taille de l'image sur la performance

Taille (pixels)	Nombre de pixels	Cycles	Fautes de cache (%)	Nombre de blocs
160x120	19200	740724	3.61	1200
352x240	84480	3326400	3.35	5280
704x480	337920	13939200	4.52	21120
1584x1080	1710720	57815140	5.84	106920

4.2 Pour l'algorithme complexe

Les résultats de performance observés sur l'algorithme complexe sont reportés dans cette partie.

4.2.1 Superficie supplémentaire et fréquence d'horloge maximum du nouveau jeu d'instructions

Le jeu d'instructions créé pour l'algorithme complexe ajoute 44816 portes au processeur standard soit une superficie additionelle d'environ 69%. La logique supplémentaire accepte en fréquence d'horloge jusqu'à 305 MHz, ce qui signifie que, comme dans le cas du jeu d'instructions pour l'algorithme simple, c'est la fréquence du processeur de base qui est considérée. Les différences que l'on observe entre les deux jeux spécialisés sont essentiellement dues à la taille des registres créés et à leur nature. Le jeu d'instructions pour l'algorithme simple crée en effet un banc de registre qui ajoute de la complexité.

4.2.2 Accélération

Dans cette partie, on regarde l'accélération obtenue lorsque le programme utilisant les instructions spécialisées est mis en oeuvre. L'accélération est présentée dans le tableau 4.6. Dans ce tableau, les accélérations de chacune des parties évoquées dans la figure 2.5 de l'algorithme sont aussi présentées. L'estimation bi-directionnelle du mouvement est réalisée par les fonctions CACULATEmotion et ASSIGNEmotion qui calculent respectivement les mouvements dans l'image et assignent à chacun des blocs de l'image à interpoler un vecteur de déplacement. Le raffinement et lissage des vecteurs trouvés sont réalisés respectivement dans les fonctions REFINEmotion et SMOOTHmotion. La

TAB. 4.6 Accélération de l'algorithme complexe

Fonctions	Cycles	%	Cycles	%	Accélération
CALCULATEmotion	417853740	94.5	2894476	28.3	144.4
ASSIGNEmotion	2525660	0.6	1992930	19.5	1.3
REFINEmotion	10505930	2.4	844670	8.2	12.4
SMOOTHmotion	4413380	1.0	526010	5.1	8.4
INTERPOLATEimage	6560100	1.5	2304100	22.5	2.9
Total	442060230	100	10235930	100	43.2

fonction INTERPOLATEimage utilise les mouvements estimés pour générer l'image interpolée. L'accélération globale obtenue est de 43.19. Les accélérations observées sur les diverses parties de l'algorithme varient de 144.4 à 1.3. Le tableau présente également la proportion de chaque partie dans le temps d'exécution total de l'algorithme. Pour ces résultats, l'algorithme travaillait avec des images de tailles 160 par 120 pixels pour des blocs 8 par 8 et un espace de recherche de 16 lors du calcul du mouvement et de 2 pendant les étapes de raffinement.

L'ensemble des tests réalisé pour l'algorithme simple (taille de l'espace de recherche et cache) n'a pas été effectué dans le cas de l'algorithme complexe car ces simulations réclament beaucoup de temps et aurait conduit à des résultats similaires puisque les deux algorithmes sont basés sur les mêmes principes.

4.3 Comparaisons de performance et de qualité

4.3.1 Génération d'images

On a mis en oeuvre deux algorithmes de MC-FRC. Le but est d'être capable de générer un grand nombre d'images afin de pouvoir traduire aisément de n'importe quel type de flux vidéo vers tout autre. La performance, ici présentée en terme de capacité

de génération d'images par seconde, doit être comparée. L'algorithme simple utilisant la version de programme ODFS-TIE peut générer plus de 470 images par seconde au format 160 par 120 pixels. Si l'on prend comme principe que l'exécution de l'algorithme est linéairement dépendant du nombre de pixels dans l'image (ceci est vrai en première approximation) alors ce programme peut générer environ 30 images par seconde dans le cas d'un flux NTSC (640 par 480 pixels soit 16 fois plus que 160 par 120). Il peut donc doubler le débit d'images d'un flux NTSC.

L'algorithme complexe peut quant à lui générer 26 images par seconde en 160 par 120 pixels soit moins de 1 par seconde pour un flux NTSC.

L'algorithme simple est donc le plus rapide mais cela n'est guère surprenant car il n'effectue aucune correction et retourne un résultat de qualité moindre que l'algorithme complexe.

4.3.2 Qualité

La qualité a déjà été discutée lors de la description des algorithmes. L'algorithme complexe est comme attendu celui qui fournit les plus beaux résultats en comparant à l'oeil nu. Cependant, ces résultats sont insuffisants. On observe toujours en effet des artéfacts dans l'image et, lorsque l'on travaille sur une séquence vidéo afin de doubler son débit, on se rend compte que ces artéfacts sont visibles lors de la diffusion de la vidéo. La conséquence est qu'un algorithme plus simple comme l'interpolation temporelle donne des résultats moins choquant pour le spectateur.

Les défauts apparaissent dans des situations diverses. Si le mouvement dans l'image est trop ample, l'algorithme ne retrouve pas la position précédente du bloc et il estime mal le mouvement de ce dernier. De même si un bloc disparaît ou est partiellement caché par un autre, l'algorithme est perdu dans ces situations.

Ces défauts sont inhérents à la classe des algorithmes utilisés. Elle suppose que les objets sont bien représentés par des blocs et que leurs mouvements sont linéaires. Les algorithmes sont donc incapables de détecter des mouvements tels que la rotation ou des situations de contraction de l'objet qui devient alors trop petit pour être "visible" par un bloc. L'erreur sur l'estimation des mouvements devient alors monnaie courante.

4.4 Discussion sur la performance et les accélérations obtenues

Ici, les résultats obtenus sont discutés et analysés. Chaque composante, chargements, calculs et contrôle, des algorithmes est regardée afin de tenter d'expliquer l'accélération globale atteinte.

4.4.1 Poids des calculs et impact de l'usage d'instructions spécialisées sur ces derniers

Dans les deux jeux d'instructions créés, les calculs de distance SAD ont été intégrés à une ou plusieurs instructions (dist pour le jeu de l'algorithme simple et sad_line pour celui de l'algorithme complexe). La version de référence du calcul de la SAD utilise deux boucles imbriquées pour réaliser ce calcul, une pour chaque coordonnée. Les instructions spécialisées permettent donc d'économiser un nombre important d'instructions standards et produisent le même résultat. Comme cela a déjà été dit, l'instruction *dist* réalise 200 opérations par appel et *sad_line* en réalise environ 180. De plus, ces deux instructions économisent davantage d'instructions en permettant de réduire le surplus dû aux boucles et aux appels de fonctions. Quand on additionne tous ces éléments, on constate qu'une seule instruction spécialisée peut remplacer plus de 200 instructions standard. Il convient de noter que cela n'entraine pas nécessairement une accélération de 200 car ces instructions prennent deux cycles pour s'effectuer (ceci afin de limiter le

chemin critique et ainsi de conserver une fréquence d'horloge élevée).

Paralléliser les calculs dans le cas de la SAD est relativement aisé. Chaque pixel peut être traité indépendamment des résultats concernant son voisin et le résultat global peut être calculé à la fin. Cependant, plus les calculs se font en parallèle, plus le besoin en données se fait important. Un algorithme, qui était déjà relativement gourmand en données, devient alors carrément affamé. Il faut donc, conjointement à l'augmentation des capacités de calcul, augmenter les capacités mémoire (stockage et bande passante). La section suivante présente comment les jeux d'instructions créés parviennent à leurs fins.

Enfin, il existe un autre cas dans lequel une instruction s'avère très performante pour remplacer et ainsi accélérer le traitement d'un code de calcul. C'est le cas de la division. Les instructions créées remplacent une fonction complexe en une simple instruction ce qui bien évidemment permet d'économiser un grand nombre d'instructions. Toutefois, l'instruction est limitée et elle n'accepte qu'un petit nombre de valeurs pour diviseur. Ceci est un bon exemple du fait que la spécialisation permet une amélioration de la performance.

4.4.2 Réutilisation des données

Comme cela a déjà été évoqué, les calculs ont été grandement parallelisés et ils ne sont plus le goulot d'étranglement de l'algorithme. Cependant, pour s'assurer que les instructions de SAD peuvent réaliser leur tâche à pleine vitesse, une quantité importante de données doit être fournie. Les algorithmes de BME sont, de façon évidente, gourmands en données et le fait d'accélérer les calculs n'améliore pas la situation. Un exemple peut être vu avec l'instruction *dist*, pour chaque appel 48 pixels de 8 bits chacun doivent être fournis ainsi qu'au moins 6 valeurs de paramètres stockées dans des registres d'états de

32 bits. Ce qui fait un total de 576 bits de données soit 18 chargements de mots de 32 bits. Pour l'instruction *sad_line*, ce nombre est de 544 bits. Afin que le processeur ne passe pas son temps à faire des chargements mémoire, il faut prendre soin de lui donner un accès rapide et large aux données.

Pour obtenir des transferts de données efficaces, la solution la plus simple et la plus viable est d'augmenter la réutilisation des données. La plupart du temps, les pixels sont réutilisés pour calculer les mouvements des blocs adjacents. Dans le cas des blocs de l'image précédente, ils partagent les trois quarts de leurs pixels avec leurs voisins. Pour augmenter la performance, il faut donc conserver les données qui sont susceptibles d'être réutilisés le plus près possible du processeur, dans les registres ou dans les caches. Pour l'algorithme simple, les registres implémentés sont capables de conserver un bloc de 4 par 4 pixels au sein du processeur. Ainsi, par exemple, le bloc courant peut être conservé dans ces registres. Comme c'est celui qui est le plus réutilisé, pour chaque calcul de distance en fait, le garder dans ces registres est la meilleure solution possible pour augmenter la réutilisation. Les instructions de chargement sont aussi créées pour augmenter la réutilisation. Comme le montre la figure 3.1, ces instructions chargent la ligne du bas du bloc et décalent les autres lignes vers le haut. On réutilise ainsi les pixels qui ont déjà été chargés. Les programmes FS-TIE2 et FS-TIE3 utilisent cette caractéristique alors que FS-TIE1 ne l'utilise pas. Comme on le voit, cela a un impact important sur la performance de ce dernier, puisqu'il est au final plus de deux fois plus lent que les deux premiers. FS-TIE1 s'appuie seulement sur la cache de données pour conserver les données près du processeur alors que FS-TIE3 utilise fortement les registres locaux comme mémoire pour la réutilisation. FS-TIE2 est une solution intermédiaire. Dans cette dernière, la boucle principale, qui traverse l'espace de recherche, réalise en moyenne 3 chargements de 32 bits pour un appel à l'instruction dist. La version de l'algorithme complexe utilisant les instructions spécialisées s'appuie fortement sur la cache. En effet, dans le cas de cet algorithme, la taille des blocs retenue empêche de les maintenir en

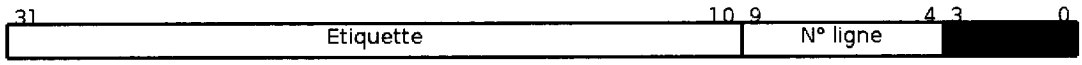


FIG. 4.2 Champs de l'adresse

entier et de façon efficace pour la réutilisation dans des registres du processeur. Ici, la solution a été d'augmenter la bande passante puisque les chargements effectués sont sur 64 bits au lieu de 32. Les lignes de cache peuvent alors être vues comme un banc de registres de second niveau avec un temps d'accès plus important.

D'une solution à l'autre, la réutilisation des données est progressivement augmentée, ce qui implique une réduction du nombre de chargements effectués par le programme. Le simulateur indique combien d'instructions de chargements ont été réalisées par le programme. Pour l'algorithme simple dans sa version de référence, un total de 14 millions de chargements est réalisé. La version de cet algorithme contenue dans le programme FS-TIE2, qui utilise donc les instructions spécialisées, effectue environ 200000 chargements. Par conséquent, le nombre de chargements a été réduit d'un facteur 70 simplement en augmentant la réutilisation des données. Pour l'algorithme complexe, on compte 58 millions de chargements dans la version de référence pour à peu près 3 millions dans celle utilisant le jeu d'instructions étendu soit une réduction de l'ordre de 20 fois.

La cache joue également un grand rôle dans la réutilisation et donc dans la performance. Sa taille et ses paramètres de fonctionnement autorisent le stockage d'un espace de recherche entier. En effet, si on reprend les valeurs des différents paramètres qui entrent en jeu (à savoir : taille de la cache, longueur des lignes, associativité, taille des images, taille des blocs et espace de recherche), on peut alors estimer le comportement de la cache lorsque l'on charge les pixels de l'espace de recherche. La cache de données est de taille 2 Ko et est configurée en ensemble associatif à 2 voies (2-way set-associative) ce qui signifie qu'elle possède deux mémoires de chacune 1024 octets. Les lignes de la cache sont larges de 128 bits, on en compte donc 64 par mémoire. La valeur 32 bit

de l'adresse est donc divisée comme le montre la figure 4.2. Les bits de 0 à 3 ne sont pas considérés par la cache car ils indiquent un octet dans une ligne. Les bits de 4 à 9 sont décodés comme le numéro de la ligne dans laquelle la donnée souhaitée pourra être trouvée. Le reste constitue l'étiquette. Pour que l'espace de recherche soit intégralement inclus dans la cache, il ne faut pas que les adresses mémoire concerné aient le même numéro de ligne. Considérons maintenant un espace de recherche de 20 par 20 débutant au pixel 0 (donc à l'adresse mémoire 0) dans le cas d'une image de taille 160 par 120 pixels. Pour recouvrir une seule ligne (20 pixels) de l'espace de recherche, on a besoin de 2 lignes de caches (2x16 pixels). La première ligne de cache est la ligne 0 et la seconde est la ligne 1. Ces deux lignes recouvrent la première ligne de l'espace de recherche. Pour savoir quelles seront les deux lignes suivantes il faut considérer l'adresse du premier pixel de ces lignes. Ce pixel se trouve à l'adresse 160 ce qui signifie donc que ce sont les lignes 10 et 11 qui recouvrent la deuxième ligne de l'espace de recherche. D'une façon générale, on peut obtenir les lignes de cache concernant une ligne de l'espace de recherche grâce à l'équation 4.1.

$$lc(n) = \frac{n \times w}{\lambda} \bmod N \tag{4.1}$$

où n est le numéro de la ligne de l'espace de recherche considérée, w la largeur de l'image, λ la longueur de la lignes de cache et N le nombre de ligne dans une mémoire de la cache. Les calculs montrent que les lignes de caches d'un espace de recherche donné sont toutes différentes tant que le rapport $\frac{w}{\lambda}$ est différent d'un diviseur où d'un multiple de N. Ici, c'est bien le cas que l'on observe avec un rapport égal à 10 et N valant 64.

Une fois que le bloc courant a été traité par l'algorithme, son voisin est à son tour traité. L'espace de recherche associé à l'ancien bloc courant et celui du bloc courant en cours partagent une grande partie de leurs pixels. Par exemple, pour un bloc de 4 par

4 et un espace de recherche s'étendant de 8 pixels dans toutes les directions à partir de la position du bloc courant (soit 20 par 20 pixels), seule une colonne de 4 pixels par 20 est nouvelle dans l'espace de recherche du bloc courant voisin soit une réutilisation d'environ 80% des données. L'autre partie de l'espace de recherche est partagée entre les deux blocs. Dans le cas du chargement du bloc précédent, et en dehors des cas de frontières, trois fois sur quatre les données sont déjà dans la cache car le bloc est large de 4 pixels, soit 32 bits, tandis que la ligne de cache contient 16 pixels sur 128 bits.

Les résultats sur la cache présentés dans le tableau 4.3 montrent que le programme de référence pour l'algorithme simple est moins sensible à la taille de la cache que le programme FS-TIE2. Ceci est principalement dû au fait que le programme de référence fait beaucoup de chargements (souvent de la même donnée) avec comme résultat que la donnée est plus souvent en cache que dans le cas des chargements pour FS-TIE2. Augmenter la taille de la cache permet d'augmenter les chances de trouver la donnée dans la cache car elle n'aura pas été évincée. On réduit donc les fautes de caches (caches misses). Concernant l'associativité, on constate que c'est la référence qui y est le plus sensible. Cela est dû au fait que cet algorithme utilise plus la pile lors des nombreux appels de fonctions. Une plus grande associativité est susceptible de corriger ce problème et donc d'accélérer plus efficacement le traitement de ce programme que celui de FS-TIE2. Toutefois augmenter l'associativité de la cache augmente la taille du contrôleur et peut également conduire à une consommation plus importante de la cache.

Les résultats pour l'ajout de délai en lecture sur mémoire prouvent que les programmes utilisant le jeu d'instruction étendu sont plus sensibles à la mémoire du fait qu'ils sont plus gourmands. Si la mémoire est lente, l'impact est plus important sur les programmes utilisant les instructions spécialisées car ils passent plus de temps à charger des données qu'à calculer. On constate aussi qu'ils utilisent pleinement la cache car même avec une latence de 100 cycles, la performance de FS-TIE2 est toujours 50 fois plus importante que celle de la référence.

4.4.3 Contrôle et limitations

Le contrôle au sein d'un algorithme peut être une tâche coûteuse en temps pour le processeur qui l'exécute. Par exemple, le programme de référence de l'algorithme simple basé sur le FS prend 67 millions de cycles pour interpoler une petite image (160 par 120 pixels). Parmi ces cycles et selon les informations retounées par le simulateur, environ 7 millions sont des cycles de surplus dus aux instructions de contrôle (boucles et conditions), soit environ 10% du total. Dans le programme FS-TIE2, il faut 780000 cycles pour compléter l'interpolation d'une image et environ 20000 sont la conséquence des surplus dus aux boucles. On a donc 3% du temps d'exécution du programme consacré au contrôle. Cette réduction de la part du contrôle est due au nouveau jeu d'instructions et plus notablement à l'instruction *dist* dont l'utilisation permet de s'affranchir d'un grand nombre de boucles.

Pour l'algorithme complexe, la part du contrôle est d'un peu plus de 4% dans la version de référence (sans les instructions spécialisées). On compte en effet 19 millions de cycles dus au contrôle pour un total de 440 millions de cycles pour générer l'image au complet. Dans le programme utilisant les instructions spécialisées, la part du contrôle est de 10%. La part du contrôle augmente donc dans ce programme. Ceci est dû au fait que cette partie est plus difficile à accélérer. On peut observer cette particularité lorsque l'on regarde les accélérations des parties de l'algorithme. Les fonctions CALCULATEmotion, REFINEmotion et SMOOTHmotion sont davantage accélérées que INTERPOLATEimage et ASSIGNEmotion. Ces dernières sont dominées par le contrôle tandis que ce sont les calculs qui dominent les premières.

4.4.4 Accélération globale : loi d'Amdhal

Comme les sections précédentes le montrent, toutes les parties de l'algorithme sont accélérées. Les calculs de distance SAD, qui contiennent du contrôle et des boucles, forment la partie majeure du temps d'exécution de l'algorithme (environ 80% dans le programme référence de l'algorithme simple). Ensuite, les chargements et écritures sont la seconde partie en terme d'importance (entre 10% et 30% dépendant de l'algorithme). Quand on utilise le jeu d'instructions spécifique pour l'algorithme simple dans le programme FS-TIE2, l'accélération des calculs SAD est estimée à environ 100 fois. Les chargements sont 70 moins nombreux. Par conséquent, quand on cherche à déterminer l'accélération globale, on peut adapter la loi d'Amdahl pour le cas où l'accélération locale réduit le temps d'exécution global. Pour le programme FS-TIE2, les chargements représentent 20% du temps d'exécution total. Cette estimation vient du fait que l'on compte 14 millions de cycles consacrés aux instructions de chargement auxquels on doit ajouter environ 340000 cycles dus aux fautes de cache (cache misses) sur un programme s'exécutant en 67 millions de cycles. Le reste du temps est consacré aux calculs. L'accélération globale est ainsi estimée dans l'équation 4.2.

$$GA = \frac{100}{\frac{80}{100} + \frac{20}{70}} \approx 92 \tag{4.2}$$

Comme on le voit le résultat théorique correspond au résultat pratique observé (voir le tableau 4.1). Cependant, cette correspondance est meilleure que celle qu'on pourrait observer de manière générale car les estimations théoriques sont entachées d'imprécision. En effet, les valeurs utilisées sont des estimations des proportions et de l'accélération de chaque partie de l'algorithme. Pour les autres programmes implantant l'algorithme simple basé sur FS et utilisant le jeu étendu, on peut noter que dans le cas de FS-TIE3 les chargements sont moins nombreux que dans FS-TIE2, ce qui permet

d'accélérer encore davantage la partie chargement. Cependant, FS-TIE3 est moins générique et fixe tous les paramètres de l'algorithme (taille des blocs et taille de l'espace de recherche). FS-TIE2 réalise moins de chargements que FS-TIE1 car il utilise mieux les propriétés de réutilisation des registres-bloc et des instructions de chargement spécialisées. Des conclusions similaires pourraient être tirées de l'analyse de l'algorithme complexe.

Dans le cas de l'algorithme simple basé sur ODFS, on peut constater que l'accélération atteinte est plus faible. Ceci est dû au fait que le programme ODFS-TIE fait plus de calculs que la référence. On peut aussi noter que la part des chargements est plus importante dans cet algorithme. L'accélération des calculs est en réalité quatre fois plus importante que celle que l'on peut observer.

On peut également appliquer la loi d'Amdahl sur les parties de l'algorithme complexe. Originellement, 94% du temps d'exécution de l'algorithme était passé dans la fonction CALCULATEmotion, les autres fonctions se partageant le temps restant. Lorsqu'on utilise le jeu d'instructions spécialisé cette fonction est accélérée d'un facteur 144 fois. Les autres fonctions sont également accélérées selon les valeurs du tableau 4.6. L'accélération globale peut donc être calculée selon l'équation 4.3.

$$GA = \frac{100}{\frac{94.5}{144.3} + \frac{0.5}{1.2} + \frac{2.3}{12.4} + \frac{1}{8.3} + \frac{1.4}{2.8} + 0.3} \approx 45 \tag{4.3}$$

Comme on le voit, les résultats sur l'accélération observés au tableau 4.6 s'expliquent bien grâce à la loi d'Amdahl.

CHAPITRE 5

ÉNERGIE ET ASIP

Le but de ce chapitre est d'évaluer et de comparer la performance énergétique d'un processeur spécialisé avec celle d'un processeur standard pour un même type d'application. Dans le contexte de ces travaux, l'algorithme utilisé est l'algorithme simple de MC-FRC. On cherche ici à caractériser plus précisement les conséquences de l'utilisation d'un jeu d'instructions spécialisé sur la consommation énergétique. Comme on l'a vu dans les chapitres précédents, la performance est significativement augmentée, il est cependant tout aussi pertinent de caractériser ce qu'il en est de la consommation énergétique. La méthodologie ainsi qu'une partie des résultats présentés ici ont fait l'objet d'une soumission à la conférence ICECS2007 [5].

5.1 Travaux antérieurs relatifs à la consommation d'énergie des ASIP

L'énergie consommée par une puce est de plus en plus au coeur des préoccupations des concepteurs en microélectronique. En effet, le marché des appareils portables a considérablement augmenté ces dernières années. Aujourd'hui, tout le monde possède un téléphone portable, un ordinateur portable ou un balladeur mp3. Comme ces appareils fonctionnent tous sur batterie, l'enjeu est donc devenu essentiel de réduire la consommation d'énergie du système tout en conservant une puissance de calcul élevée. Récemment, on assiste même à l'augmentation des capacités de ces machines. La toute dernière génération de téléphones portables permet aux utilisateurs de téléphoner, d'écouter de la musique, de prendre des photos, de regarder des films ou encore de filmer. Toutes ces capacités réclament une grande puissance de calcul mais sans augmentation de la taille

de la batterie qui conduirait à un appareil plus lourd et plus encombrant. Certes les technologies pour fabriquer des batteries font des progrès, mais ceux-ci restent insuffisants en regard de la demande. Les concepteurs s'attachent donc à réduire la consommation de leur système.

Comme on le voit, le problème est de créer un système capable d'offrir des performances importantes tout en ayant une consommation sobre. Les ASIPs ont prouvé qu'ils étaient capables d'offrir de grandes capacités de calculs, il convient donc de s'intéresser ici à leur consommation énergétique.

Les ASIPs sont souvent présentés comme une technique à mi-chemin entre les processeurs standards et les ASICs. Ils possèdent la généralité des premiers et ont des performances comparables aux seconds. En revanche, les ASICs ont une consommation moindre. Pour réduire cette différence, de nombreux travaux ont portés sur la réduction de l'énergie consommée par l'ASIP. Ces travaux se sont intéressés à de nombreuses techniques comme le portillonage de l'horloge (clock gating), l'amélioration de l'arbre d'horloge (clock tree), le réglage plus adapté des ressources du processeur, la modification de l'encodage des instructions ou encore l'ajout d'un mode sommeil [13]. Toutes ces techniques ont prouvé leur efficacité et une large partie d'entre elles sont d'ores et déjà intégrées au processeur Xtensa LX. D'autres travaux ont suggéré l'utilisation de plusieurs ASIPs pour en remplacer un [17]. L'idée est de réduire la taille des ASIPs. Les résultats montrent que plus petits et plus spécifiques, ils consomment moins qu'un ASIPs plus générique, donc plus gros et moins efficace.

Bien que ces méthodes permettent de réduire en moyenne l'énergie consommée, elles ne permettent pas de connaître à l'avance la consommation du système. En effet, pour les ASIPs comme pour les processeurs standards, il est difficile de connaître la consommation réelle à l'avance. Elle dépend certes de l'architecture interne du processeur mais aussi en grande partie du programme qui est exécuté. En conséquence, de

nombreux travaux se sont attachés à développer des modèles pour estimer rapidement et facilement la consommation qu'engendrera un programme tournant sur le processeur (qu'il soit standard, configurable ou spécialisé) [12, 24]. Cette estimation se fait à l'aide de compilateurs C et de simulateurs possédant une base de connaissances sur la consommation engendrée par toutes les instructions du processeur. Les modèles montrent comment établir cette base de connaissance et comment l'utiliser. Il est intéressant de constater que [24] s'intéresse plus particulièrement aux processeurs destinés à être embarqués sur un FPGA. La méthodologie présentée est assez proche de celle qui a été utilisée dans le cadre de nos travaux.

Enfin, une troisième catégorie de travaux s'intéresse aux architectures possibles pour les ASIPs et à leurs conséquences sur les performances et la consommation énergétique [1]. Le type d'architecture retenue est un élément important sur les caractéristiques de consommation d'un processeur.

Pour conclure cette revue de littérature, il est à noter qu'aucun travail, à notre connaissance, ne s'intéresse à la comparaison de la consommation d'énergie entre un processeur standard et un processeur spécialisé lorsqu'ils réalisent un même algorithme.

5.2 Méthodologie

Dans cette section, nous présentons la méthodologie retenue pour obtenir et comparer les consommations énergétiques d'un processeur standard et d'un processeur spécialisé.

Le but de cette comparaison est de caractériser la performance énergétique du jeu d'instructions étendu. Comme on l'a vu, la spécialisation des instructions permet d'accélérer le traitement de l'algorithme d'un facteur de plus de 90 fois. Cela a été réalisé

en augmentant la superficie du processeur. Dans le cadre de cette étude, on s'intéresse maintenant au ratio de consommation entre le processeur spécialisé et le processeur standard. Les résultats obtenus ne sont donc pas une indication sur la consommation précise de chaque processeur mais plutôt une indication sur les performances énergétiques que peut attendre le concepteur qui réalise un jeu d'instruction par rapport à l'accélération obtenue et au coût en surface.

La méthodologie est basée sur une mise en oeuvre sur FPGA du processeur Xtensa. Le code source HDL du processeur étant inaccessible, c'est donc un FPGA Virtex II de Xilinx[29] qui sert de base pour cette méthodologie. Les résultats restent toutefois valides dans le cadre de la conception d'une puce spécialisée (ASIC). En effet, on ne s'intéresse ici qu'aux ratios de consommation d'énergie entre un processeur étendu et la version standard. Ces ratios étant établis à partir des ratios de l'activité interne des processeurs, ces ratios restent identiques quelle que soit la technologie (ASIC ou FPGA) utilisée. Par ailleurs, la méthodologie présentée ici pourrait être aisément adaptée à une implantation ASIC du processeur en remplacant les composants du FPGA par des cellules normalisées.

En premier lieu, une liste des interconnexions (netlist) du processeur (sous la forme d'un fichier NGO propre aux outils Xilinx) est récupéré en utilisant l'outil XPG de création de processeur. Le processeur peut être créé en lui ajoutant les instructions spécialisées ce qui permet de créer deux versions du processeur : la version dite *standard* qui ne possède pas de jeu d'instructions spécifique et la version dite *étendue* qui possède le jeu d'instructions supplémentaire construit pour l'algorithme simple. Un certain nombre de fichiers tab :Acc=E9l=E9ration-algocomperilog accompagnent le fichier de liste des interconnexions. Ils concernent l'interface de bus et l'enveloppe Verilog nécessaires pour les étapes de synthèse qui suivent. À cette étape, on récupère également les compilateurs C et les simulateurs rattachés à chaque processeur.

Les fichiers Verilog et la liste des interconnexions NGO obtenus sont ensuite synthétisés à l'aide des outils Xilinx ISE pour obtenir un fichier de liste des interconnexions NGD. À ce stade, le fichier NGD contient l'intégralité du FPGA. On peut alors traduire cette liste d'interconnexions en une description VHDL du FPGA avec l'outil ngd2vhdl de Xilinx. À la fin de cette étape, une description au niveau porte du FPGA contenant le processeur et son interface de bus est disponible.

L'étape suivante consiste à créer un banc d'essai pour le FPGA. Normalement, ce banc d'essai devrait représenter l'intégralité de la carte de prototypage. Dans ce cas, un simple bus de 32 bits auquel sont connectées deux mémoires : une mémoire ROM qui contient le code d'initialisation du processeur et une mémoire RAM qui contient le programme et ses données, est créé. Les mémoires sont créées pour être suffisamment grandes pour contenir l'intégralité du programme. Elles possèdent également la capacité de s'initialiser à partir du contenu d'un fichier afin de faciliter le chargement du programme en mémoire.

Une fois que le système complet est modélisé dans des fichiers VHDL, il est prêt à être simulé. On doit alors compiler le code source C pour extraire le contenu des mémoires du programme obtenu. Pour ce faire, on utilise le compilateur xt-xcc de Tensilica qui retourne le programme compilé. Ce dernier est désassemblé à l'aide de xt-objdump qui retourne un fichier texte décrivant la suite d'instructions réalisant le programme, leur encodage dans le langage machine ainsi que leur adresse dans l'espace mémoire du processeur. Un script analyse alors le contenu de ce fichier pour en extraire le contenu de chaque mémoire qu'il décrit dans un fichier texte où les données sont représentées en valeur hexadécimale avec une ligne pour chaque valeur de 32 bits, la première ligne représentant la valeur située à l'adresse 0 de la mémoire concernée.

La simulation est ensuite lancée sur le simulateur Modelsim [21]. Pour enregistrer l'activité interne du FPGA, on utilise l'outil de surveillance de transition intégré à Mo-

delsim. Cet outil compte le nombre de commutations (toggle) à 1 et à 0 qui se produisent pour chaque signal à l'intérieur du FPGA. Afin de ne pas s'encombrer de la séquence d'initialisation dans les mesures, on ne fait démarrer le décompte qu'une fois la fonction *main()* du programme atteinte. Le décompte s'arrête lorsque la fin du programme est atteinte. La simulation retourne un fichier texte qui contient une liste de l'intégralité des signaux du FPGA auxquels sont associés le nombre de transitions à 1 et à 0.

Une fois que le nombre de transitions pour chaque signal est connu, on peut calculer le taux d'activité de chacun d'entre eux. Ici, on ne considère que les transitions de 1 à 0 pour le calcul du taux d'activité, en effet les nombres de transition à 1 et à 0 sont identiques à un près aux nombres de transitions de 0 à 1 car les signaux ne sont jamais indéfinis. On peut avec les taux d'activité des signaux déduire la puissance consommée dans le FPGA. Pour ce faire, on utilise l'outil de Xilinx intitulé Power Estimator Worksheet Webtool [30]. Cet outil utilise les taux d'activité moyens des sorties de chacun des composants du FPGA (CLB, blocs RAM, etc.) pour estimer la puissance consommée. Afin d'obtenir ces taux, on utilise des scripts qui analysent le code VHDL du FPGA et associent pour chaque composant le signal de sortie correspondant. La valeur moyenne est ensuite calculée pour chaque type de composant. Pour les blocs RAM, les signaux d'écriture et d'activation sont considérés. Une fois ces valeurs obtenues, on peut compléter le formulaire de l'outil d'estimation de puissance en ligne et obtenir la puissance consommée par le processeur lorsqu'il fait rouler le programme. La puissance considérée est la puissance dynamique. Elle tient compte du routage dont l'influence peut être précisée comme faible, moyenne ou élevée. Comme l'influence du routage est toujours négative (i.e. la puissance consommée augmente avec le routage), ces trois cas seront considérés comme les meilleur, normal et pire cas de consommation, respectivement. La consommation d'énergie est déduite en intégrant la puissance consommée sur le temps d'exécution du programme.

Toutefois, comme l'influence du routage peut être importante, il faut procéder à

une comparaison entre les valeurs obtenues pour chacun des processeurs avec précaution. Les comparaisons doivent donc être faites entre chacun des cas pour établir le ratio minimum entre les deux processeurs, le standard et le spécialisé.

5.3 Résultats

Dans cette section nous présentons les résultats obtenus pour les deux processeurs grâce à la méthodologie évoquée dans le paragraphe précédent. Il faut se rappeller que les processeurs sont tous basés sur un Xtensa LX. Ils possèdent une cache de donnée de 2 Ko configurée en ensemble associatif à deux voies et une cache d'instruction de 1 Ko à accès direct. Ils peuvent adresser un bus externe de 32 bits via une interface mémoire de 128 bits et un adaptateur. Le processeur étendu implante les instructions spécialisées créées pour l'algorithme simple.

Le processeur standard utilise 25% de la logique du FPGA soit 17353 LUTs ainsi que 39 blockRAMs. Le processeur étendu a besoin de 44% avec 29828 LUTs et 39 blockRAMs. On observe donc bien la même augmentation en taille que celle obtenue en synthèse sur le FPGA (72% d'augmentation dans le cas du FPGA contre 79% pour l'implémentation ASIC, voir 4.1.1). Les blockRAMs sont utilisés par la cache dans les deux cas. Comme cette dernière ne change pas de taille entre les deux processeurs, il est normal que le même nombre de blockRAM soit utilisé.

Chaque processeur a donc été implanté sur FPGA et a exécuté le programme de MC-FRC correspondant (la référence FS pour le processeur standard et le programme FS-TIE2 pour le processeur spécialisé). Cependant, comme cet algorithme est gourmand en calculs et que les simulations requises prennent du temps, il est impossible de simuler l'intégralité du processus. Des versions réduites des programmes ont donc été utilisées. Des simulations ISS, bien plus rapides, ont également été faites afin d'obtenir des sta-

TAB. 5.1 Résultats de consommation (mW) pour le processeur standard

nombre de blocs	1	4	8
puissance (meilleur cas)	265	268	268
puissance (cas normal)	288	292	292
puissance (pire cas)	359	365	366
cycles d'horloge	65925	246738	491431
faute de cache de données(%)	0.75	0.99	0.65
faute de cache d'instructions(%)	0.36	0.09	0.05

tistiques sur la cache ainsi que sur le nombre probable de cycles qui seront requis pour compléter l'algorithme réduit. Toutefois, bien que le nombre de cycles requis soit une indication, il faut noter que le simulateur ISS et le FPGA n'ont pas le même bus. Le bus FPGA étant plus lent. Cela n'a pas d'incidence sur les résultats de consommation car le bus est identique pour les deux processeurs, celui étendu et le standard et seul le rapport de consommation entre les processeurs nous intéresse.

Comme évoqué, les simulations FPGA utilisent un programme réduit pour des considérations de temps. Sur le processeur étendu, estimer le mouvement pour un bloc est facile, il est donc possible aisément de calculer jusqu'à 40 blocs dans un temps de simulation relativement raisonnable. En revanche, sur le processeur standard la tâche est plus ardue et un simple bloc requiert énormément de temps (environ 6 heures de calculs sur un processeur Pentium 4 à 3 GHz). En conséquence, sur ce processeur, les simulations n'ont pas été plus loin que 8 blocs. Les tableaux 5.1 et 5.2 présentent les résultats obtenus pour le processeur standard et le processeur étendu respectivement.

Ces tableaux indiquent les puissances moyennes estimées (en mW) lorsque les processeurs exécutent des versions réduites à un certain nombre de blocs (1 à 40). Trois cas de consommation, reliés aux différentes influences du routage, sont rapportés. Le nombre de cycles nécessaires pour compléter l'algorithme sur la simulation FPGA est également indiqué, ainsi que les statistiques sur les caches (fautes de cache) fournies par

TAB. 5.2 Résultats de consommation (mW) pour le processeur spécialisé

nombre de blocs	1	4	8	40
puissance (meilleur cas)	320	356	368	399
puissance (cas normal)	346	392	409	448
puissance (pire cas)	426	502	529	595
cycles d'horloge	1950	4140	7021	28744
faute de cache de données(%)	29	12	7.75	3.76
faute de cache d'instructions(%)	6.4	2.6	1.48	0.35

les simulations ISS. Il est intéressant de noter ici l'impact du taux d'échec des caches sur la consommation et sur la performance. Si l'on regarde le cas du processeur étendu, on s'aperçoit que le programme réduit à 1 bloc a une consommation bien plus faible que celui à 40 (environ 2 fois moins importante). En revanche, le traitement d'un bloc est au moins 2 fois plus long (1950 contre environ 719). Cela est dû aux taux d'échec des caches qui ralentissent l'exécution et diminuent l'activité au sein du processeur. En effet, comme la donnée n'est pas présente dans la cache il faut donc aller la chercher en mémoire externe. Pendant ce temps, le processeur attend et ne fait rien, ce qui le ralentit mais réduit sa consommation moyenne.

Pour que la consommation engendrée par le programme réduit soit considérée comme un bon indicateur de celle du processeur exécutant l'algorithme dans sa globalité, il faut donc considérer un algorithme réduit qui possède des statistiques de caches similaires à celles obtenues pour l'image complète. Dans le cas du processeur standard travaillant sur l'image au complet, le taux d'échec de la cache de données est de 0.40%, celui de la cache d'instruction de 0%. L'algorithme réduit à 4 blocs est donc un indicateur valide pour la consommation du processeur standard lors du traitement de l'image au complet. Sur le processeur étendu travaillant sur l'image en intégralité, le taux d'échec pour la cache de données est de 3.61% et celui de la cache d'instructions de 0.01%. C'est donc l'algorithme réduit à 40 blocs qui représente la valeur la plus proche de celle que

TAB. 5.3 Ratios de consommation d'énergie

	standard		
étendu	pire	normal	meilleur
pire	53.5	43.6	40.4
normal	69.7	56.9	52.7
meilleur	77.6	63.3	58.6

l'on obtiendrait sur l'image complète. Ce sont donc les valeurs de ces deux programmes réduits qui seront comparées.

Afin d'effectuer une comparaison valide, il convient d'ajouter, à la consommation de ces deux processeurs, la consommation de puissance créée par les entrées/sorties. Sur le processeur standard travaillant avec le programme à 4 blocs, cette consommation supplémentaire est de 32mW dans tous les cas. Sur le processeur spécialisé avec le programme à 40 blocs, elle est de 36mW.

Maintenant que les consommations de puissance sont bien établies, on peut comparer les consommations d'énergie. Le programme tournant sur le processeur standard calcule 4 blocs en 246738 cycles d'horloges soit 61685 cycles par bloc. Celui sur le processeur étendu calcule 40 blocs en 28744 cycles soit 719 cycles par bloc. L'accélération est donc de plus de 85. On peut donc à l'aide de cette valeur calculer les ratios de consommation. Le tableau 5.3 présente les résultats obtenus.

Comme ces résultats ne permettent pas d'avoir un idée précise de l'activité interne des processeurs, on s'intéresse également à l'activité des signaux. La figure 5.1 présente les histogrammes de l'activité des signaux au sein des deux processeurs. Cette courbe permet de voir l'activité moyenne ainsi que la répartition de signaux autour de cette moyenne.

Les figures 5.2 et 5.3 présentent les histogrammes de l'intégralité des signaux

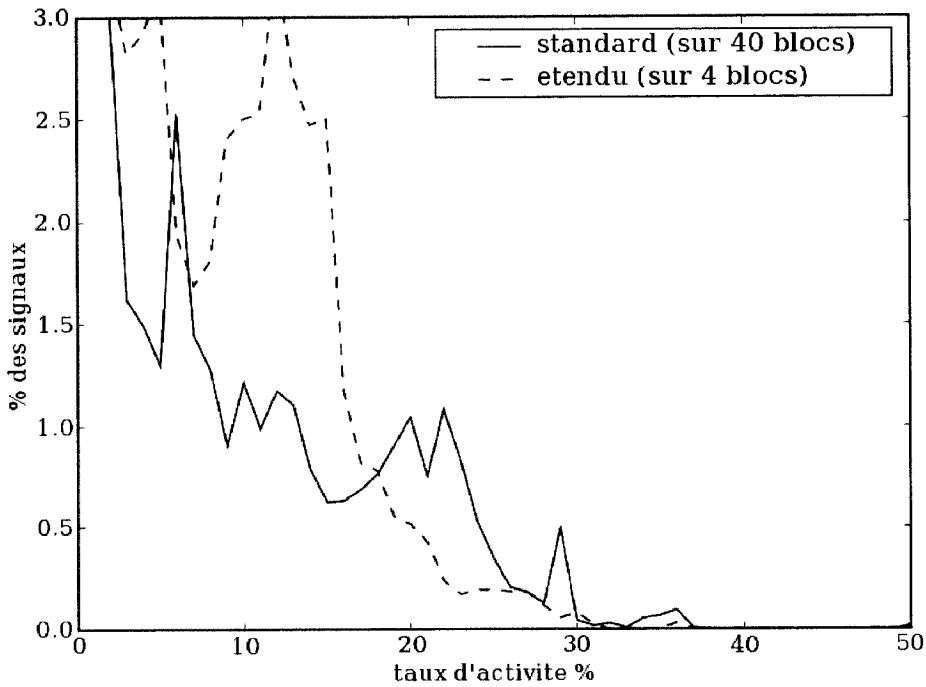


FIG. 5.1 Histogramme de l'activité des signaux à l'intérieur des deux processeurs

de chacun des processeurs ainsi que ceux des signaux consacrés aux registres et aux unités de traitement (ALU). Ces signaux sont importants car ils représentent la partie du processeur qui fait les calculs. De plus, il est bon de noter que, dans le cas du processeur étendu, une large partie de la logique ajoutée est consacrée à ces éléments.

Les conclusions que permettent d'établir ces figures sont présentées dans la section suivante.

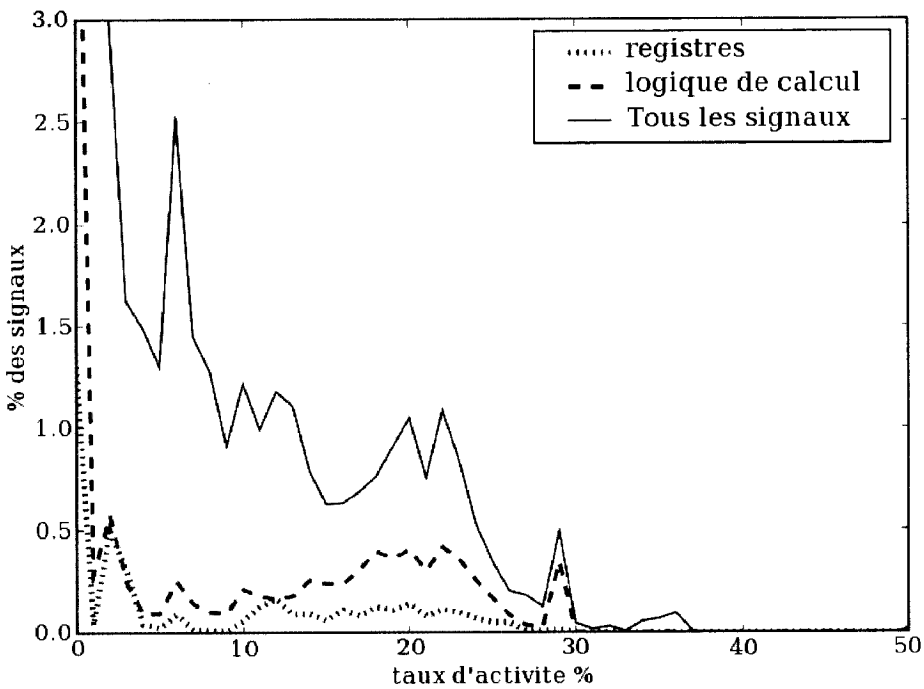


FIG. 5.2 Histogramme de l'activité des registres et de unités de traitement dans le processeur standard

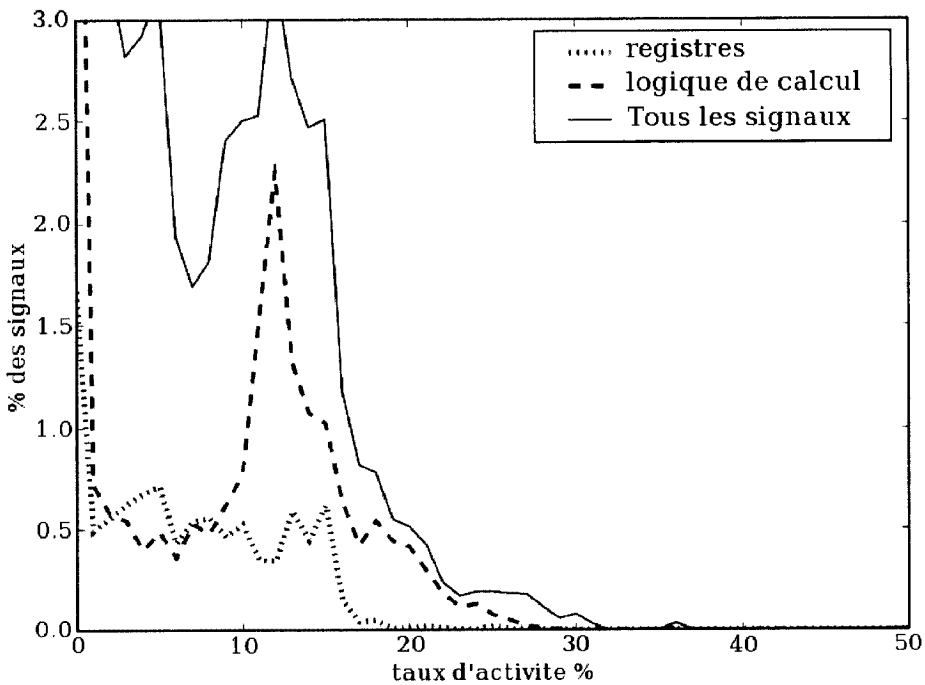


FIG. 5.3 Histogramme de l'activité des registres et unités de traitement au sein du processeur spécialisé

5.4 Discussion

Dans cette section, les résultats obtenus sont analysés. Tout d'abord, il convient de regarder plus en détails l'architecture interne du processeur Xtensa LX. Ce processeur est basé sur 5 modules aux fonctions propres : 1) l'unité de chargement et sauvegarde qui gère la cache de données et procède aux chargements et aux sauvegardes réclamés par les instructions correspondantes dans le code exécuté par le processeur ; 2) L'unité de prise en charge des instructions (instruction fetch) et le compteur de programme (program counter) qui maintient à jour le compteur de programme et charge les instructions en fonction de son contenu ; 3) L'interface de bus qui permet de connecter le processeur à un bus de données ; 4) L'unité de contrôle qui gère le bon déroulement de l'exécution et finalement 5) l'unité de calculs dans laquelle les instructions sont décodées et exécutées sur les données stockées dans les registres. Cette dernière unité constitue les derniers étages de pipeline du processeur et contient le décodeur, les registres et les ALU.

Lorsque l'on compare les activités des deux processeurs (étendu et standard), il ressort que seule l'activité au sein de l'unité de calcul est modifiée de façon significative entre les deux processeurs. Comme cela a été dit précédemment, la logique ajoutée sur le processeur spécialisée est en grande partie consacrée à cette unité, ce résultat n'est donc pas surprenant. Dans cette unité, on trouve le décodeur qui est également augmenté afin de gérer les nouvelles instructions mais son activité varie peu. Seuls les registres et les unités de traitement voient leur activité profondement modifiée par le jeu d'instructions étendu.

Si l'on regarde plus en détails ces éléments dans les deux processeurs, on s'aperçoit des choses suivantes : dans le processeur étendu le nombre de signaux concernant les registres est de 5650 alors que 12860 signaux font partie des unités de traitements. L'activité moyenne de ces signaux se situe aux alentours de 12% où on peut observer

un pic sur l'histogramme de la figure 5.3. Dans le processeur standard, les nombres de signaux pour les registres et pour les unités de traitement sont respectivement de 1307 et de 3936. Ces signaux sont assez bien distribués entre 0 et 30% avec un léger plateau entre 20 et 30% (figure 5.2). La logique supplémentaire est donc 2.5 fois plus importante que la logique de base puisque le processeur étendu compte 18510 signaux dans cette partie contre 5243 dans le processeur standard (la logique de base est aussi présente dans le processeur étendu). Cependant, l'introduction de cette logique a permis de réduire le taux d'activité qui passe de 20% à 12%. Cette différence d'activité est partiellement due au fait que la logique de base dans le processeur étendu est moins sollicitée donc son faible taux d'activité diminue la moyenne. De plus, on remarquera que les instructions spécialisées qui réalisent un nombre important de calculs sont gourmandes en données. On se retrouve donc dans une situation où le processeur procède à davantage de chargements que de calculs. Comme les instructions de chargement n'engendrent pas une activité importante dans l'unité de calcul comparativement à des instructions de calcul comme *add*, il en résulte une diminution du taux d'activité dans cette unité (l'activité est alors concentrée dans l'unité de chargement et de sauvegarde). Le processeur étendu passe donc beaucoup de temps à attendre des données de la cache. L'activité est donc réduite dans les étages d'exécution du pipeline alors que, sur le processeur standard, ces étages travaillent énormément pour réaliser un nombre important d'instructions de calcul qui sont la partie la plus importante de l'algorithme. À la fin, on constate que même si la logique a plus que triplée, la consommation de puissance n'augmente que de 50% (pour les comparaisons avec cas similaires).

Comme le processeur étendu accélère le temps de traitement d'un facteur 85, la puissance consommée supplémentaire est effacée par l'accélération temporelle. La consommation d'énergie qui en découle est donc essentiellement rattachée à l'accélération du temps de calcul. Ainsi, même si l'on compare le meilleur cas pour le processeur standard avec le pire cas pour le processeur étendu ce dernier est toujours 40 fois plus

économe.

Pour conclure, on pourra noter que le facteur d'économie d'énergie peut être résumé comme la division de l'accélération du temps de calcul par le rapport des superficies entre les deux processeurs. Toutefois, cette relation est basée sur des conjectures. Afin de la vérifier deux cas de tests supplémentaires ont été implantés. Les résultats sont présentés dans la section suivante.

5.5 Validation de l'équation simplifiée

Les résultats sur la consommation d'énergie obtenus avec le processeur spécialisé dans le cas des algorithmes de MC-FRC pourraient, comme on l'a déjà dit, être plus rapidement estimés grâce à une équation simple reliant le rapport de consommation en énergie (E) entre un processeur spécialisé et un processeur standard, l'accélération de performance (A) que le premier permet par rapport au second et le rapport des superficies (S) entre les deux processeurs. L'équation 5.1 qui suppose que la cadence d'horloge est identique pour les deux processeurs permettrait alors d'estimer rapidement le gain sur la consommation d'énergie obtenu par un processeur spécialisé à l'aide de chiffres plus aisément accessibles que sont l'accélération et le rapport de superficie.

$$E = \frac{A}{S} \tag{5.1}$$

Afin de valider ou d'invalider cette équation, deux algorithmes ont été implantés, accélérés et caractérisés. Le premier algorithme est un calcul de la suite de Fibonacci. Le second algorithme est un algorithme de pré-encodage utilisé dans la norme JPEG2000. Ces deux algorithmes ont l'avantage de présenter des cas différents de celui du MC-FRC. De plus, la suite de Fibonacci est un cas extrême (pire cas), tel que décrit plus loin, ce qui

la rend particulièrement intéressante puisqu'elle permet de borner l'erreur de l'équation.

5.5.1 La suite de Fibonacci

Le calcul de la suite de Fibonacci est simple comme nous le montre l'équation 5.2, Le $n^{\text{ième}}$ terme de la suite est égale à la somme du $n-1^{\text{ième}}$ terme avec le $n-2^{\text{ième}}$ terme. Les deux premiers termes de la suite peuvent prendre n'importe quelle valeur. Ici, on prendra 1 pour les deux.

$$u_n = u_{n-1} + u_{n-2} \tag{5.2}$$

Du point de vue algorithmique, le calcul de la suite présente plusieurs caractéristiques. En premier lieu, comme on le constate, il n'est pas gourmand en donnée. Le processeur ne doit conserver en mémoire qu'un nombre très faible de valeurs (3 au maximum et 2 avec une optimisation facile). Ces valeurs sont donc conservées dans les registres internes. La deuxième caractéristique importante est que les calculs sont simples puisqu'il ne s'agit que d'additions mais ces calculs sont très séquentiels. Normalement on ne peut pas en effet calculer le $n^{\text{ième}}$ terme sans avoir les valeurs du $n-1^{\text{ième}}$ et du $n-2^{\text{ième}}$. Ceci explique pourquoi le calcul de cette suite a été retenu, en effet c'est algorithmiquement l'opposé du MC-FRC.

L'implantation logicielle du calcul de la suite de Fibonacci se fait donc avec la boucle suivante :

```
for(i=0;i<NITER;i++){
  tmp=f[1];
  f[1]=f[1]+f[0];
  f[0]=tmp;
}
```

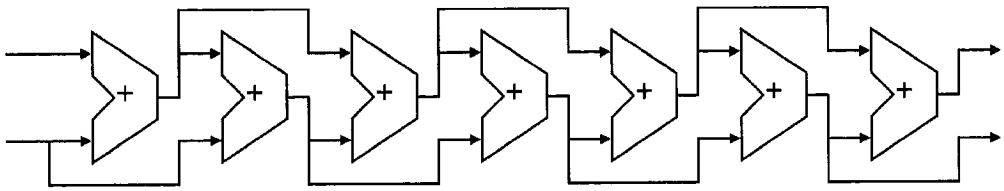


FIG. 5.4 Instruction pour calculer la suite de Fibonacci

Pour accélérer ces calculs, une seule instruction a été codée. Elle réalise sept additions en chaîne comme le montre la figure 5.4. Cette instruction permet d'accélérer, en théorie, de sept fois le calcul de la suite.

Afin de caractériser la consommation d'énergie et d'obtenir toutes les valeurs de l'équation, la méthodologie présentée en 5.2 a été appliquée dans le cas de cet algorithme. Pour la version non accélérée, on reprend un processeur identique à celui de l'algorithme de MC-FRC. Ce processeur requièrt donc 17353 LUTs ainsi que 39 blockRAMs. Le processeur intégrant l'instruction spécialisée requièrt quant à lui 17516 LUTs et 39 blockRAMs. Comme on peut le voir, l'ajout de l'instruction spécialisée n'augmente pas de façon importante la logique du processeur (moins de 1%). On fait calculer à l'algorithme la $8000^{\text{ième}}$ itération de la suite. Sur le processeur standard, ce calcul prend 7014 cycles d'horloges contre 1071 cycles pour le processeur spécialisé. Les consommations sont présentées dans le tableau 5.4. On peut constater d'ores et déjà sur ce tableau que le processeur spécialisé consomme moins en puissance. Le tableau 5.5 montre les ratios d'énergie pour la suite de Fibonacci. On peut voir que l'hypothèse de l'équation simplifiée est invalidée par cet exemple. En effet, l'équation aurait donné un ratio d'énergie proche de la valeur de l'accélération puisque l'augmentation en superficie est faible alors que le ratio d'énergie vaut presque le double de l'accélération.

Cette différence s'explique par plusieurs facteurs. Les figures 5.5 et 5.6 montrent la répartition de l'activité au sein des deux processeurs. Comme on peut le constater, le processeur standard a un taux d'activité plus important. On peut voir deux maximums,

TAB. 5.4 Consommation de puissance (mW) pour la suite de Fibonacci

	meilleur cas	cas standard	pire cas
Standard	296	327	399
Spécialisé	177	181	195

TAB. 5.5 Ratios de consommation d'énergie pour la suite de Fibonacci

	standard		
étendu	pire	normal	meilleur
pire	15.3	12.5	11.4
normal	16.5	13.5	12.2
meilleur	16.9	13.8	12.5

l'un à environ 24% d'activité et l'autre à environ 50%. Ce sont des activités relativement élevée. Le processeur spécialisé ne présente quant à lui qu'un seul pic à environ 23% d'activité. Le nombre de signaux concernés par cette activité est aussi réduit puisque le maximum est bien inférieur à celui du processeur standard. Dans ce cas-ci, comme l'ajout de la logique est faible, les deux processeurs ont une quantité de signaux internes comparable, ce qui n'était pas le cas avec l'algorithme de MC-FRC. La figure 5.6 montre l'activité des registres et des unités arithmétiques et logiques (ici l'additionneur pour le processeur standard et la chaîne d'additionneurs de l'instruction spécialisée). On constate que, dans les deux processeurs, la première pointe d'activité est la conséquence des calculs et des registres. On note toutefois que la part des registres est considérablement réduite, voire disparait, dans le cas du processeur spécialisé. Ceci provient du fait qu'un grand nombre de résultats intermédiaires n'ont plus besoin d'être enregistrés.

Cette figure n'explique toutefois pas le deuxième maximum d'activité dans le processeur standard. Ce maximum est dû à l'activité dans le module de compteur de programme et de prise en charge des instructions comme le montre la figure 5.7. Cela est dû à une particularité de l'architecture du Xtensa. En effet, ce dernier possède une ins-

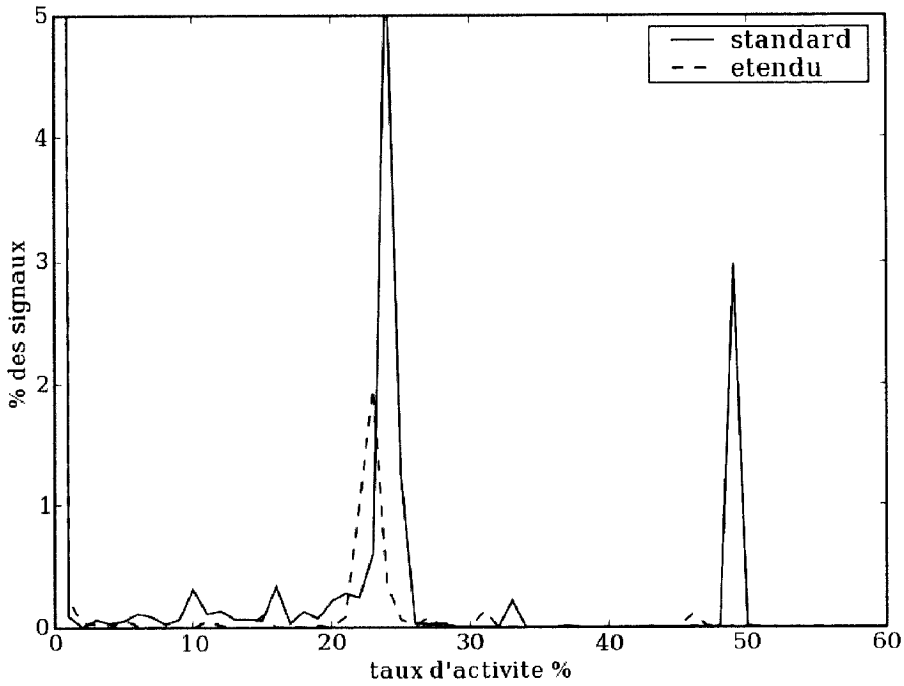


FIG. 5.5 Histogramme de l'activité dans les processeurs réalisant la suite de Fibonacci

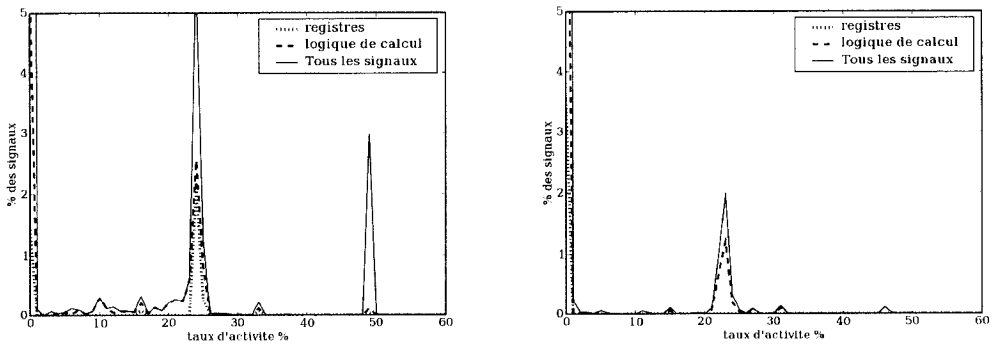


FIG. 5.6 Activité dans les registres et les ALU pour la suite de Fibonacci (standard à gauche)

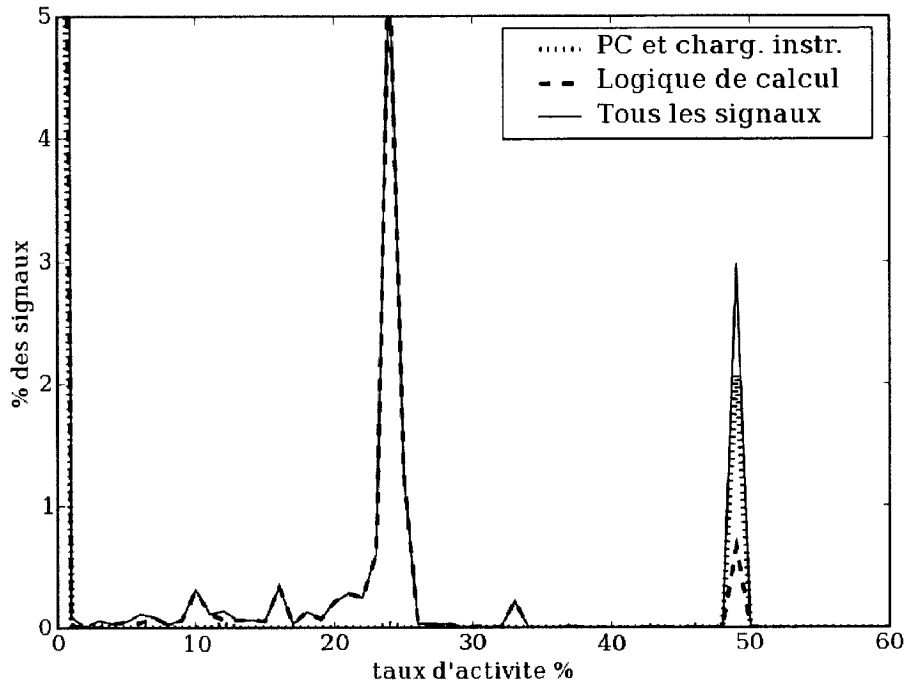


FIG. 5.7 Activité dans le module de chargement d'instructions et de la sémantique (logique de calcul) dans le processeur standard pour la suite de Fibonacci

truction permettant de répéter autant que souhaité et de façon automatique un bout de code du programme sans avoir à gérer des instructions de fin de boucle. Dans le cas de la suite de Fibonacci, cela est très utile pour le processeur spécialisé qui n'a qu'à répéter un grand nombre de fois son instruction spécialisée. Comme il n'y a qu'une seule instruction et que c'est tout le temps le même encodage cela résulte dans une activité nulle dans l'unité d'Instruction Fetch. Dans le cas du processeur standard, il n'y a aussi qu'une seule instruction, add, mais cette dernière est utilisée de façon différente. En effet, le compilateur transforme le code C à l'intérieur de la boucle présentée plus haut en un code assembleur suivant :

```
add a2, a3, a2

add a3, a2, a3
```

Comme on le voit la fonctionnalité est préservée, la boucle n'est répétée que 4000 fois au lieu des 8000 prévues dans le code C, mais le résultat est identique et plus rapide. Cependant, l'encodage de ces instructions est différent et est fixé par les outils (0x232a pour la première et 0x323a pour la seconde) et cela a pour conséquence de faire basculer la valeur d'un certain nombre de signaux à chaque nouveau cycle d'où le maximum à 50%. Le calcul de la suite de Fibonacci invalide donc l'équation mais il faut retenir que ce cas est particulier. De plus, les résultats s'appuient sur une particularité de l'architecture du Xtensa. La conséquence de l'utilisation de l'instruction de boucle auto répétée est que toutes les parties du processeur qui ne sont pas utilisées sont éteintes. L'utilisation de l'instruction spécialisée a pour conséquence réduire significativement (voire éteindre) l'activité dans tous les modules hormis ceux des calculs (instruction spécialisée) de cette instruction. C'est donc une situation que l'on peut considérer comme une forme de cas extrême pratique.

TAB. 5.6 Consommation de puissance (mW) pour le *Magnitude Coding*

	meilleur cas	cas standard	pire cas
Standard	290	308	359
Spécialisé	294	309	359

5.5.2 *Magnitude Coder* du JPEG2000

Le *Magnitude Coding* est une étape de la compression JPEG2000 qui permet d'extraire des données des couples D-CX où D est un bit encodé et CX est le contexte qu'on lui attache. Cette étape permet ensuite de compresser de façon très efficace avec un algorithme d'encodage arithmétique binaire, le MQ-Coder. Le *Magnitude Coding* est un algorithmes extrêmement séquentiel et il est basé sur des arbres de décisions complexes et séquentiels. De plus, les traitements se font sur des bits et non sur des mots de données classiques (8 ou 32 bits).

Dans ce chapitre, on ne présente que les résultats obtenus pour la consommation. Le jeu d'instructions créé étant relativement conséquent et hors de propos ici. On retiendra que 9 instructions ont été créées. Le processeur spécialisé nécessite 20198 LUTs contre 17353 pour le standard, soit une augmentation en surface d'environ 16%. L'algorithme est réalisé en 1570 cycles sur le processeur spécialisé et 16151 sur le processeur standard. L'accélération est donc de 10.29. Le tableau 5.6 montre la consommation de puissance des deux processeurs lorsqu'ils travaillent sur l'algorithme. Le tableau 5.7 présente les ratios d'énergie observés qui sont centrés autour de la valeur 10.3.

5.5.3 Conclusion sur l'équation simplifiée

Afin de conclure sur la validité de l'équation, on rassemble dans le tableau 5.8 les valeurs attendus par l'équation simplifiée et celles obtenues en pratique pour les

TAB. 5.7 Ratios de consommation d'énergie pour le *Magnitude Coding*

	standard		
étendu	pire	normal	meilleur
pire	10.29	8.85	8.42
normal	11.99	10.32	9.82
meilleur	12.73	10.96	10.43

TAB. 5.8 Résultat attendus et observés sur l'énergie

Valeur	MC-FRC	Fibonacci	Magnitude Coding
Attendue	50	6.5	9
Observée	56	12	10.5
Erreur	+12%	+85%	+17%

trois algorithmes considérés. La valeur expérimentale est la moyenne de la diagonale du tableau contenant les ratios d'énergie, i.e. la moyenne des cas où les deux processeurs sont comparés de façon équitable pour le placement et routage.

L'erreur d'approximation réalisée par l'équation simplifiée, qui peut paraître importante, est relativement faible en regard de sa simplicité. On constate de plus que la pire erreur, observée pour un cas très particulier, ne dépasse pas les 100% même pour la suite de Fibonacci qui, comme on l'a expliqué, est une forme de cas extrême pratique. Enfin, on peut s'apercevoir que l'économie d'énergie prédite par l'équation simplifiée est toujours en deçà de celle qu'on observe. L'équation est donc pessimiste, ce qui vaut généralement mieux que d'être trop optimiste. Donc, on obtient facilement un estimé pessimiste avec une borne sur l'erreur précise et serrée, ce qui particulièrement intéressant lorsqu'on conçoit un système électronique.

Bien entendu ces résultats ont été obtenus sur un nombre restreint d'exemples et dans des situations simples. Des études supplémentaires seraient requises pour valider

ou invalider cette équation dans des situations plus complexes. Par exemple, on pourrait considérerke cas d'un processeur étendu par deux jeux d'intructions spécialisés, chacun pour une application différente, et offrant des accélérations différentes pour ces deux applications. Ainsi le jeu d'instructions de la première application n'influencerait pas l'accélération obtenue pour la seconde mais changerait la taille du processeur étendu. De même, des situations où la fréquence d'horloge est différente entre les deux processeurs devraient être étudiées pour s'assurer que l'équation reste valide.

CHAPITRE 6

TRAVAUX ANNEXES ET EXTENSIONS FUTURES

Dans ce chapitre, on présente les travaux et réflexions découlant des résultats présentés précedemment. Dans un premier temps, on présente les travaux effectués pour créer un système permettant de connecter un réseau sur puce, ici un tissu d'interconnexions, à un processeur Xtensa étendu avec un jeu d'instructions lui permettant d'accélérer un traitement vidéo. Le but étant d'avoir un infrastructure permettant de réaliser des systèmes multi-processeurs. Ces travaux permettent aussi de valider les résultats obtenus dans un environnement plus complet. La dernière section de ce chapitre présente une réflexion sur les algorithmes de MC-FRC. En effet, la conclusion de ces travaux sur ces algorithmes porte à penser que leur qualité ramenée à leur coût en calculs impose de trouver de meilleurs algorithmes que ceux basés sur l'estimation de mouvement par bloc.

6.1 ASIP et Réseau sur puce

La figure 6.1 montre le système complet d'un processeur Xtensa connecté à un réseau sur puce (Tissu d'interconnexions - Switch Fabric). Ce système a été créé pour réaliser un algorithme de MC-FRC sur un flux vidéo entrant via l'interface stream IN et dont le résultat sort par l'interface stream OUT. L'arbitre gère le bon déroulement de l'algorithme au sein du système.

Ce système a été simulé à l'aide de simulations SystemC. Le processeur Xtensa est étendu à l'aide du jeu d'instruction spécialisé pour l'algorithme de MC-FRC simple. Les modules stream IN et stream OUT se chargent respectivement de faire entrer et sortir

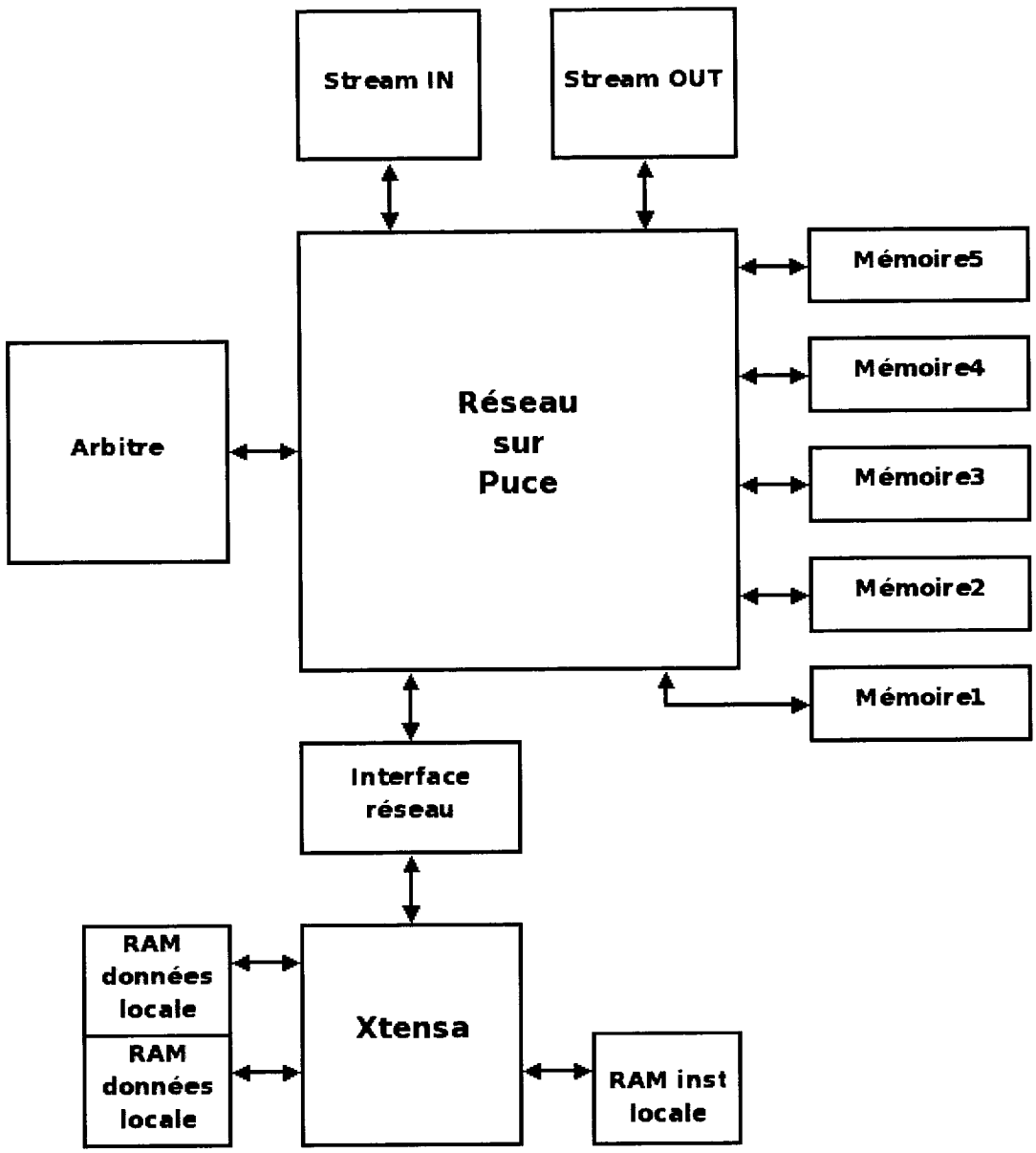


FIG. 6.1 Schéma du système Réseau sur Puce - Xtensa

un flux vidéo basse résolution (160 par 120 pixels). Le débit d'image est doublé grâce à l'algorithme simple de MC-FRC implanté grâce au programme FS-TIE2.

Les résultats préliminaires montrent qu'il faut un peu moins de 800000 cycles au processeur spécialisé pour créer une image intermédiaire entre le moment où il en reçoit l'ordre par l'arbitre et le moment où il est prêt pour recommencer l'opération sur une autre image. Le système complet a été capable d'ajouter 10 images à un flux en entrée en 7 millions de cycles.

Ces 7 millions de cycles sont à comparer avec les résultats du programme FS-TIE2. Ces résultats montraient qu'il était possible de réaliser le calcul d'une image en moins d'un million de cycles. On constate donc des performances similaires lorsque l'on passe au système complet avec même une légère amélioration. Toutefois, de nombreuses optimisations restent à produire. Notamment, en ce qui concerne l'interface entre le processeur et le réseau sur puce. En effet, le DMA qu'elle contient est extrèmement basique et souffre d'un manque de souplesse qui pourra être pénalisant pour des applications futures. De plus, ces résultats sont obtenus avec des simulations qui ne portent pas une grande attention à la taille des paquets qui transitent par le réseau. La taille de ces derniers est également un facteur important de la performance du réseau. Dans le cas évoqué ici, il n'y a qu'un unique processeur. Le réseau sur puce n'est donc pas très sollicité et dans une utilisation plus avancée (multi-processeur) la performance du réseau sera un facteur très important.

6.2 Optimisation des algorithmes de compensation de mouvement

Un des grands credos des algorithmes complexes de MC-FRC basés sur du Block Matching est qu'il faut pouvoir trouver les mouvements réels. Ainsi, les algorithmes placent beaucoup d'efforts pour corriger les vecteurs de mouvements trouvés afin d'éli-

miner des erreurs de prédiction. L'algorithme complexe présenté dans ce mémoire n'échappe pas à cette règle puisqu'une étape (l'étape de lissage) est consacrée à la correction des vecteurs de mouvement prédits. Cependant, de telles techniques restent, malgré tout, insuffisantes puisqu'elles ne solutionnent pas des problèmes intrinsèques à la classe d'algorithme utilisée. La limitation sur le type de mouvement est un exemple frappant de ce genre de limitation. Un autre problème est que les "objets" dont on considère le mouvement doivent être correctement représentés par des blocs, ou par des ensembles de blocs.

Ainsi, afin d'améliorer ces algorithmes deux axes doivent être considérés. Le premier concerne la possibilité de considérer d'autres catégories de mouvement. Le second porte sur la détection des "vrais" objets de l'image afin de travailler sur leur mouvement plutôt que sur celui des blocs.

CONCLUSION

Le présent mémoire a présenté les travaux réalisés au cours de ma maîtrise. Ces travaux portaient sur les algorithmes d'augmentation de taux de trames par compensation de mouvement (Motion Compensated Frame Rate Conversion - MC-FRC). Différentes classes existent parmis ces algorithmes, les travaux présentés ici se sont intéressés aux algorithmes basés sur la division en blocs des images. Ces derniers sont couramment utilisés dans les codecs vidéo tels que la famille MPEG. Deux algorithmes ont été implementés afin de déterminer leurs caractéristiques. Le premier, considéré comme simple, n'est en fait que le noyau de ce type d'algorithme et il permet de se faire une bonne idée des enjeux découlant de leur utilisation. Le second est une version plus complexe qui fait appel à plusieurs techniques pour obtenir un meilleur résultat. Dans les deux cas, les algorithmes réclament une grande puissance de calculs. Le déroulement des algorithmes a donc été accéléré à travers la conception de jeux d'instructions spécialisés. Ces derniers ont permis des accélérations de plus de 100 fois pour l'algorithme simple, avec 116 pour le programme FS-TIE3, et de plus de 43 fois pour l'algorithme complexe. Ces résultats s'expliquent par la mise en oeuvre de solutions basées sur la paralllélisation et la réutilisation de données. De telles accélérations ont ensuite soulevé la question de l'efficacité énergétique des processeurs spécialisés. Une méthodologie a donc été mise en place afin de comparer la consommation d'énergie d'un processeur standard avec celle d'un processeur spécialisé. Les résultats ont montré que le processeur spécialisé était plus de 50 fois plus efficace en termes d'energie que le processeur standard dans le cas de l'algorithme simple. Une règle simple, liant l'accélération de performance, le rapport des surfaces entre les deux processeurs et le ratio d'énergie entre ces derniers aurait pu de façon aisée et rapide prédire ce résultat. Afin de prouver cette règle, trois cas de test ont été implantés et soumis à la même méthodologie. Les résultats ont montrés que la règle, malgré sa simplicité, peut être utilisée pour prédire de façon pessimiste mais assez

précise le gain en énergie offert par le processeur spécialisé. Les travaux se sont conclus sur des simulations d'un système complet centré autour d'un processeur spécialisé et d'un réseau sur puce de type tissu d'interconnexions. Les performances d'un tel système se sont montrées identiques voire supérieures. Toutefois ces premiers résultats ont été obtenus dans des situations très simplifiées et il convient également de garder à l'esprit que de nombreuses optimisations restent à produire pour avoir de véritables chiffres de performances.

Les travaux futures pourraient traiter de plusieurs aspects : 1) La création et l'implantation d'algorithmes de MC-FRC plus performants et de meilleure qualité qui seraient ensuite accélérés par des processeurs spécialisés ; 2) Le déploiement de ces processeurs dans des architectures orientées multi-processeur et réseau sur puces ; 3) Enfin des simulations supplémentaires permettraient de mieux couvrir l'étendue des possibilités explorées et d'appronfondir la validation du modèle de l'équation simplifiée concernant la consommation d'énergie.

RÉFÉRENCES

[1] ATAK, O., ATALAR, A., ARIKAN, E., ISHEBABI, H., KAMMLER, D., ASCHEID, G., MEYR, H., NICOLA, M., AND MASERA, G. "Design of application specific processors for the cached fft algorithm,". In *Acoustics, Speech and Signal Processing, 2006. ICASSP 2006 Proceedings. 2006 IEEE International Conference on* (2006), vol. 3, pp. III–1028–III–1031.

[2] ATASU, K., POZZI, L., AND IENNE, P. "Automatic application-specific instruction-set extensions under microarchitectural constraints,". *International Journal of PArallel Programming 31* (2003), 411–428.

[3] BEUCHER, N., BÉLANGER, N., SAVARIA, Y., AND BOIS, G. "Motion compensated frame rate conversion using a specialized instruction set processor,". In *Signal Processing Systems Design and Implementation, 2006 IEEE Workshop on* (2006), pp. 130–135.

[4] BEUCHER, N., BÉLANGER, N., SAVARIA, Y., AND BOIS, G. "High acceleration for video application using specialized instruction set based on parallelism and data reuse,". *Soumis à Journal of DSP Technologies* (2007).

[5] BEUCHER, N., BÉLANGER, N., SAVARIA, Y., AND BOIS, G. "A methodology to evaluate the energy efficiency of application specific processors,". In *Soumis à Electronics, Circuits and Systems, 2007 14th IEEE Conference on* (2007).

[6] CASTAGNO, R., HAAVISTO, P., AND RAMPONI, G. "A method for motion adaptive frame rate up-conversion,". *Circuits and Systems for Video Technology, IEEE Transactions on 6* (1996), 436–446.

[7] CHAU, L.-P., AND JING, X. "Efficient three-step search algorithm for block motion estimation in video coding,". In *Acoustics, Speech, and Signal Processing, 2003. Proceedings. (ICASSP '03). 2003 IEEE International Conference on* (2003), vol. 3, pp. III–421–4 vol.3.

[8] CHEN, M. J., CHEN, L. G., AND CHIUEH, T. D. "One-dimensional full search motion estimation algorithm for video coding,". *Circuits and Systems for Video Technology, IEEE Transactions on 4* (1994), 504–509.

[9] CHOI, B.-T., LEE, S.-H., AND KO, S.-J. "New frame rate up-conversion using bi-directional motion estimation,". *Consumer Electronics, IEEE Transactions on 46* (2000), 603–609.

[10] COWARE INC. [Online]. http ://www.coware.com.

[11] DUFAUX, F., AND MOSCHENI, F. "Motion estimation techniques for digital tv : a review and a new contribution,". *Proceedings of the IEEE 83* (1995), 858–876.

[12] FEI, Y., RAVI, S., RAGHUNATHAN, A., AND JHA, N. K. "A hybrid energy-estimation technique for extensible processors,". *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 23* (2004), 652–664.

[13] GLOKLER, T., AND MEYR, H. "Power reduction for asips : a case study,". In *Signal Processing Systems, 2001 IEEE Workshop on* (2001), pp. 235–246.

[14] HA, T., LEE, S., AND KIM, J. "Motion compensated frame rate conversion by overlapped block-based motion estimation algorithm,". In *Consumer Electronics, 2004 IEEE International Symposium on* (2004), pp. 345–350.

[15] HILMAN, K., PARK, H. W., AND KIM, Y. "Using motion-compensated frame-rate conversion for the correction of 3 :2 pulldown artifacts in video sequences,". *Circuits and Systems for Video Technology, IEEE Transactions on 10* (2000), 869–877.

[16] HUANG, A.-M., AND NGUYEN, T. "A novel motion compensated frame interpolation based on block-merging and residual energy,". In *Multimedia Signal Processing, 2006 IEEE 8th Workshop on* (2006), pp. 395–398.

[17] KALYANARAMAN, V., MUELLER, M., SIMON, S., STEINERT, M., AND GRYSKA, H. "Power reduction of asips by distributing the workload on seve-

ral asip-instances,". In *Circuit Theory and Design, 2005. Proceedings of the 2005 European Conference on* (2005), vol. 3, pp. III/457–III/460 vol. 3.

[18] KIM, S. D., LEE, J. H., HYUN, C. J., AND SUNWOO, M. H. "Asip approach for implementation of h.264/avc,". In *Design Automation, 2006. Asia and South Pacific Conference on* (2006), p. 7 pp.

[19] LYNCH, W. E. "Bidirectional motion estimation based on p frame motion vectors and area overlap,". In *Acoustics, Speech, and Signal Processing, 1992. ICASSP-92., 1992 IEEE International Conference on* (1992), vol. 3, pp. 445–448 vol.3.

[20] MBAYE, M., LEBEL, D., BELANGER, N., SAVARIA, Y., AND PIERRE, S. "Design exploration with an application-specific instruction-set processor for ela deinterlacing,". In *Circuits and Systems, 2006. ISCAS 2006. Proceedings. 2006 IEEE International Symposium on* (2006), p. 4 pp.

[21] MENTOR GRAPHICS. "Modelsim, mentor graphics,". [Online]. http ://www.mentor.com.

[22] MOHAMMADZADEH, N., HESSABI, S., AND GOUDARZI, M. "Software implementation of mpeg2 decoder on an asip jpeg processor,". In *Microelectronics, 2005. ICM 2005. The 17th International Conference on* (2005), pp. 310–315.

[23] MOSTAFIZUR, M., MOZUMDAR, R., KARURI, K., CHATTOPADHYAY, A., KRAEMER, S., SCHARWAECHTER, H., MEYR, H., ASCHEID, G., AND LEUPERS, R. "Instruction set customization of application specific processors for network processing : a case study,". In *Application-Specific Systems, Architecture Processors, 2005. ASAP 2005. 16th IEEE International Conference on* (2005), pp. 154–160.

[24] OU, J., AND PRASANNA, V. K. "Rapid energy estimation of computations on fpga based soft processors,". In *SOC Conference, 2004. Proceedings. IEEE International* (2004), pp. 285–288.

[25] PEYMANDOUST, A., POZZI, L., IENNE, P., AND DE MICHELLI, G. "Automatic instruction set extension and utilization for embedded processors,". In *Application-Specific Systems, Architectures, and Processors, 2003 IEEE International Conference on* (2003), pp. 108–118.

[26] PO, L.-M., AND MA, W.-C. "A novel four-step search algorithm for fast block motion estimation,". *Circuits and Systems for Video Technology, IEEE Transactions on 6* (1996), 313–317.

[27] TENSILICA INC. "Xtensa LX,". [Online]. http ://www.tensilica.com.

[28] TENSILICA INC. *Xtensa LX Microprocessor Data Book For Xtensa LX Processor Cores*. 2006.

[29] XILINX INC. [Online]. http ://www.xilinx.com.

[30] XILINX INC. "Xilinx virtex-II power tool version 8.1.01,". [Online]. http ://www.origin.xilinx.com/cgi-bin/power_tool/power_Virtex2.

[31] YANG, S., WOLF, W., AND VIJAYKRISHNAN, N. "Power and performance analysis of motion estimation based on hardware and software realizations,". *Computers, IEEE Transactions on 54* (2005), 714–726.